- The system supports the ability to have the user view and agree to license terms upon subscription of a license
- The system can deliver applications to any client capable of connecting to the internet over a standard HTTP connection
- The system can service customers behind corporate and personal firewalls
- The system can be integrated to work with any existing billing system and database, although a consulting and customization fee will be charged for integration with any database other than SQL server and any billing system other than ?
- The system provides an external API that makes it easy to interface with standard billing solutions
- The system provides capabilities for reviewing billing information at the global, account and user levels
- The system provides user management (create, edit, remove, enable, disable, remove, add to group, remove from group) capabilities including user groups
- The system provides account management (create, edit, remove, enable, disable) capabilities that are integrated with the accounts in the billing system (a 1-to-1 relationship between billing accounts or sub-accounts and system accounts)
- The system provides application management (create, edit, enable, disable, remove, add to group, remove from group) capabilities including application groups
- The system provides subscription and un-subscription capabilities that allow a user group access to an application group under a supported license model
- The system provides capabilities for automatically upgrading an application upon next use, for allowing a client to optionally upgrade on each subsequent use or for informing a client that an upgrade is available for subscription on each subsequent use

**2.2 Localization**

- The server components and web interface components must be able to operate across the different languages supported by Windows, although only American English menus, panels, messages, help and documentation will be initially made available

**2.3 Usability**

- The system must not require any scheduled or unscheduled down time
- The system provides appropriate help panels to the system administrator

**2.4 Reliability**

- Given sufficient hardware resources and an appropriate configuration, the system must be able to tolerate the failure of any one server either hosting the application or the software license management service without interrupting the service of any current or future clients

- The system should tolerate single failures in which a server or client transmits bogus messages or fails to transmit expected messages

### 2.5 Performance

- The system's overall price/performance makes it profitable to deliver low-cost client applications at volume levels
- The system must be able to operate satisfactorily with an average bandwidth of 256Kb/s or more
- The system must be able to operate satisfactorily with an average latency of 0.25 seconds or less
- The system allows encryption to be enabled or disabled either globally or on a domain basis to improve system performance inside protected LAN environments
- The system uses bandwidth discovery to determine whether compression is beneficial when sending massages between the client and the server

### 2.6 Scalability

- A fully-functional, non-redundant server system can be set up and configured to run on a single server machine
- The system must be scalable, given sufficient machine resources, to support any number of clients and certified applications
- The system can make effective use of the servers available to serve applications and license tokens without the need for constant administrative supervision and management

### 2.7 Security

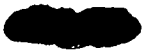
- The server must be able to operate behind an industry-standard firewall solution
- The system must withstand DOS (Denial Of Service) attacks with only localized degradations in service
- The system must allow diagnostic modes to be locked out or disabled to ensure that no outside party, including OTI, can log on and administer the server components
- The system allows for different levels of account, billing, user and subscription access so that the various capabilities can be allowed or denied on a user or user group basis
- The system logs all application usage activity to a specific user and client so that appropriate security and usage audits can be performed

### 2.8 Portability

- The server hardware platform does not limit the platforms that applications can be delivered to from that server

### 2.9 Maintainability





- The system must incorporate troubleshooting facilities to ease the burden of identifying, isolating and resolving operational issues
- The server components must be able to log all abnormal activity and the administrator should be able to select which activity should be logged and which ignored
- The server supports SNMP queries so that it can be integrated with a system management solution like Openview
- The server must be easily configured so that alarms and alerts can be defined when important events occur
- New applications and application upgrades can be installed without interrupting the current use of that application or any other applications by any client
- It must be easy to add, remove, enable and disable application and license servers without affecting the rest of the system, requiring the system to be shut down or affecting application delivery to any current or future client (provided that sufficient alternative servers are available to maintain operations)

## *3.0 Client User Requirements*

The following requirements are primarily driven by a client user needs and expectations on the system.

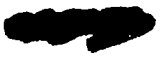
### 3.1 Functionality

- The system supports multiple applications running on the same client, including multiple instances of the same application (within reasonable limitations of the operating system, the client hardware and the applications)
- The system allows a single client to be simultaneously connected to multiple application service providers
- The system provides the capability to view and edit billing, account, subscription and user information from a connected client
- The system supports user roaming so that a user can access their applications from any machine that has installed the client and can access an application service provider over the network
- The system allows user data to be saved on the local machine
- The system allows applications to print to any local and network printers accessible to the client machine
- The system allows any set of applications to interact with each other via OLE automation, COM and signals regardless of whether the applications are installed locally or delivered by the system
- The system can be deactivated and reactivated by the user on demand
- The system can be set up to work through a proxy server

### 3.2 Localization

- The client components must be able to operate across the different languages supported by Windows, although only American English menus, panels, messages, help and documentation will be initially made available





### 3.3 Usability

- The average interactive response time of an application delivered by the system will not exceed 110% that which would be experienced by the user were running a locally-installed version of the application
- The system client components are easily downloaded and installed
- The system client components are easily uninstalled
- The system detects the presence of a local version of the application being installed and warns the user that their local version will become unavailable while the system is operational
- The system allows a pre-existing local version of the application to become available when the system's version of the application is uninstalled
- The system does not require a reboot of the client whenever a new application is installed or uninstalled
- The system removes all application-related traces from the client system when an application is uninstalled
- The system must be able install itself and operate on a client with only 16MB of disk space available, although the system may request that the user free up and reserve more disk space to improve the performance of the system
- The user can, through an advanced options menu, manage the size of the local system cache if they choose to override the system default behavior
- The system provides the user access to information regarding cache usage, bandwidth usage and connectivity status
- The user must be able to control which license to use when it can obtain a license from more than one connected application service provider
- The system provides appropriate help panels to the user
- The system allows a user to indicate that they do not wish to be informed of an optional upgrade again for a particular application and respects their choice on that matter
- The system quickly releases non-simultaneous use licenses after their use so that the same user can use the license from another client
- The system release non-simultaneous use licenses within a short span of time after a client that was using such a license crashes or is unexpectedly shut down
- A client is given sufficient notification of an impending license expiration to save their work and take appropriate action to renew or extend the license
- Upon an expiration of a license, the application is halted and the user given the ability to renew the license. The application is never terminated by the client components unless the user gives their consent to an actual termination

### 3.4 Reliability

- The client components can run indefinitely as they consume only a managed amount of system resources
- The system must deliver code and data to the client identical to that it would have were the application residing locally
- The client component can "heal" itself from deleted client dlls

- The system detects and recovers from garbled network messages although at potentially reduced performance

### 3.5 Performance

- The client must make a run/no run license decision quickly for any application execution request
- The system collects profile information locally so that future accesses are tailored to local usage patterns

### 3.6 Scalability

- There must be no system-imposed limits on how many applications can be subscribed, installed or executed simultaneously

### 3.7 Security

- All subscription, financial and credit card transactions are performed securely making it computationally infeasible for a third party to obtain any such information
- It must be computationally infeasible for the system to be coerced by a third-party into delivering a Trojan
- It must be computationally infeasible for a third party to determine which application a client is executing
- You must be able to specify that you do not wish your application usage information to be made available to any third party

### 3.8 Portability

- The system automatically delivers the appropriate version of a subscribed and installed application to the client provided that an appropriate certified version of the application is available at the application service provider's site for the client hardware and operating system configuration

### 3.9 Maintainability

- The client components should not require any user administration
- The system client components should be upgradeable without requiring a re-installation or re-subscription of existing applications

readme.txt arai [redacted]

This directory /docs is for eStream design documentation only.

This directory is reserved for high level documents such as the eStream high level requirements and design documents.

The subdirectories "client," "server," and "builder" are for documents specific to those pieces of eStream. These directories will be used to contain component-wide documents, such as the server component framework and the eStream set format document. Each directory will contain subdirectories for each sub-component. The names of the subcomponent directories will be the abbreviation or acronym for the component, in all lower case. Some client components will be ecm (cache manager), efsd (file system driver), epf (prefetching and fetching), and cni (client network interface).

Another subdirectory of "documents" will be "eng". This will include all company-wide engineering documents, such as coding guidelines and design document templates.

Document and directory names will contain no spaces. (These interoperate poorly with non-Microsoft platforms.) Files will contain the the name of the component followed by the acronym for the document type, separated by a hyphen '-'. Recognized document types are currently:
LLD - Low level design
REQ - requirements
HLD - High level design
SM - strawman

Thus some documents would be
CacheManger-LLD.doc (low level design for the cache manager)
PrefetcherFetcher-SM.doc (profiler/prefetcher component strawman)

## Estream 1.0 Planning Document

[redacted]

### Low-Level Design Status/Plan

| Sub Components | Owner | LLD Design Doc completed | LLD review Compited | Estimates for Impl | Impl and Unit Test Completed |
|---|---|---|---|---|---|
| **Content** | | | | | |
| Install Monitor | Sanjay | Done | Done | 3 wk | |
| Builder GUI | Sanjay | Done | Done | 1 wk | |
| FSRFD (Drivers) | Sanjay | Done | Done | 2 wk | |
| AppInstallBlk structure | David | Done | Not needed | | |
| Profiler | David | Done | Done | 2 wk | |
| File Access Monitor | David | Done | Done | 1 wk | |
| Packager | David | Done | Done | 1 wk | |
| **eStream distribution** | **Bob** | **8/31/2000** | **Status TBD** | **TBD** | |
| **Server Group** | | | | | |
| Web Server | Bhaven | Done | Done | 8 wk | |
| Monitor | Mike | Done | Done | 4 wk | |
| SLiM Server | Amit | Done | Done | 2 wk | |
| App Server | Sameer | Done | Done | 4 wk | |
| Admin UI | Bhaven | TBD | TBD | TBD | |
| End User UI | Bhaven | TBD | TBD | TBD | |
| Common Server Components | Mike | Done | Done | 3 wk | |
| Messaging | Sameer | Done | Done | 3 wk | |
| Threads Package | Sameer | No Document | | 1 wk | |
| **Security Design** | **Igor/Amit** | **Not Done** | **Not Done** | **TBD** | |
| **Client Group** | | | | | |
| Cache Prefetching | Anne | Done | Done | 1 wk | |
| LSM + Plug in | Anne | Done | Done | 1 wk | |
| Client UI | Anne | Done | Done | 1 wk | |
| Client Installer | Anne | Done | Done | 1 wk | |
| Start Client | Anne | Done | Done | 1 wk | |
| Application Install Mgr | Nick | Done | Done | TBD | |
| Piracy | Nick | Done | Done | TBD | |
| File Spoofer | Curt | Done | Done | 1 wk | |
| eStream File System | Curt | Done | Done | 8 wk | |
| **NoCluster Driver** | **Curt** | **Status TBD** | **Starus TBD** | **2 days** | |
| eStream Cache Manager | Dan | Done | Done | 8 wk | |
| Client Network Interface | Dan | Done | Done | 2 wk | |

[redacted]

### Implementation Plan

### Milestones

[redacted]

**ECM (RAM disk cache) and EFSD executes a local "himom" executable**
**Photoshop is installed locally and successfully executed from estream sets and appInstallblk produced by builder**
**App Server and EMS integrated to copy "himom" executable using a dummy client**
**App Server, EMS and CNI integrated to copy "himom" executable from "himom" estream sets**
**office is installed locally and successfully executed from estream sets and appInstallblk produced by builder**
**App Server, EMS, ENI, ECM and EFSD integrated to run "himom" from estream sets on server**
**Following applications built and tested with local installation**

- *Adobe Premier*
- *Macromedia Director and Shockwave*
- *Corel Suite*
- *Lotus Suite*

[redacted]

Photoshop is installed by AIM and executed from estream sets on App server

- *No Subscription*
- *No License Management*
- *RAM cache for ECM*
- *Installation of Photoshop using AIM*

[redacted]

Photoshop is installed by AIM and executed from estream sets on App server

- *No Slim Server*
- *Disk based cache for ECM*

*Estream includes initial prefetched pages and these pages are prefetched during installation*
*Fully functional estream bits (includes initial prefetched pages)*
*Client software is run as a service*
*App Server is started by Monitor*
*Admin UI to stop and start app Server*
*Application subscription from web server*
*installation on client after subscription*

[redacted]
Testing environment is setup (configuration of 3 servers and one client)
Photoshop runs with the following additional functionality

Leads for milestone: Amit and Nick
*Slim Server*
*http protocol*
*CNI supports uique message ids for NAD*
*Fully functional LSM*
*Real Accesstokens*
*Uninstall applications*
*Anti-Piracy support*
*AppServer and SlimServer fail-over*
*File spoofing*

[redacted]
Clean builds by integration (George) (Raj will drive this)

[redacted]
Office is running with full functionality

*Restruturing of client so it can be started at boot time*
*Performance tuning*
*Improve robustness*
*application upgrade*
*Crash resiliency*
*All software purified and memory leaks eliminated*
*(May be) Applets for monitoring server components*

[redacted]
**Office is removed from desktop of at least one person and reinstalled using estream**

[redacted]
Code Freeze

**Engineer**

[redacted]

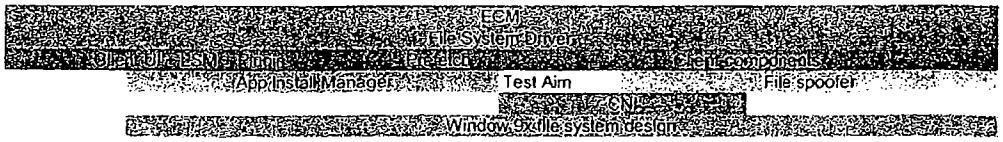
**Server**
Sameer
Mike
Bhaven
Amit
Jae Jung
Chungying Chu


**Builder**

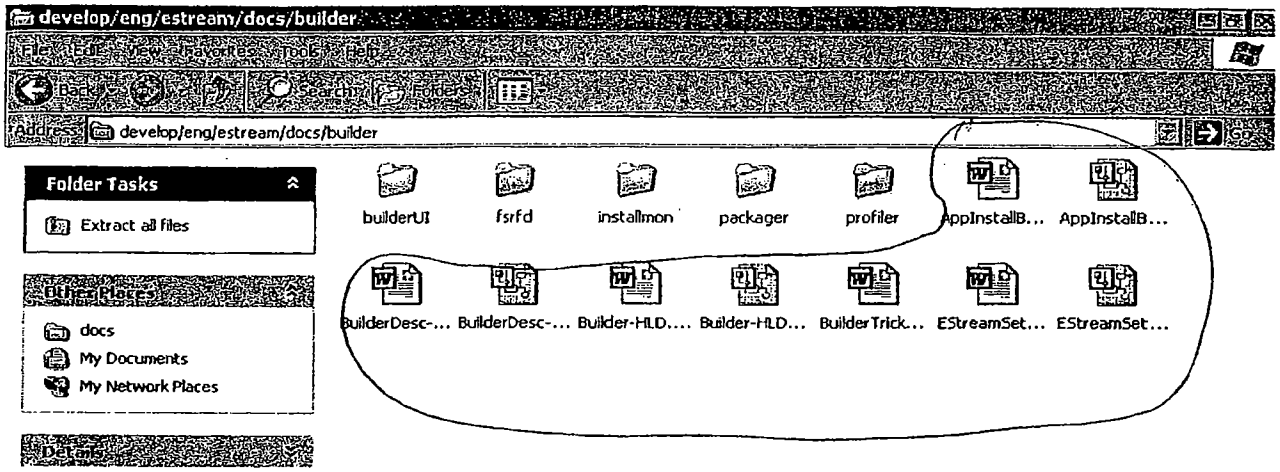
David
Bob
Sanjay


**Client**

Dan
Curt
Anne
Nick
Raj
Ameet



Builder







[redacted]

# eStream AppInstallBlock Low Level Design

*Sanjay Pujare and David Lin*
*Version 0.5*
[redacted]

## Functionality

The AppInstallBlock is a block of code and data associated with a particular application. This AppInstallBlock contains the information needed to by the eStream client to 'initialize' the client machine before the eStream application is used for the first time. It also contains optional profiling data for increasing the runtime performance of that eStream application.

The AppInstallBlock is created offline by the eStream Builder program. First of all, the Builder monitors the installation process of a local version of the application installation program and records changes to the system. This includes any environment variables added or removed from the system, and any files added or modified in the system directories. Files added to the application specific directory is not recorded in the AppInstallBlock to reduce the amount of time needed to send the AppInstallBlock to the eStream client. Secondly, the Builder profiles the application to obtain the list of critical pages needed to run the application initially and an initial page reference sequence of the pages accessed during a sample run of the application. The AppInstallBlock contains an optional application-specific initialization code. This code is needed when the default initialization procedure is insufficient to setup the local machine environment for that particular application.

The AppInstallBlock and the runtime data are packaged into the eStream Set by the Builder and then uploaded to the application server. After the eStream client subscribed to an application and before the application is run for the first time, the AppInstallBlock is send by the server to the client. The eStream client invokes the default initialization procedure and the optional application-specific initialization code. Together, the default and the application-specific initialization procedure process the data in the AppInstallBlock to make the machine ready for eStreaming that particular application.

## Data type definitions

The AppInstallBlock is divided into the following sections: header section, variable section, file section, profile section, prefetch section, comment section, and code section. The header section contains general information about the AppInstallBlock. The information includes the total byte size and an index table containing size and offset into other sections. In Windows version, the variable section consists of two registry tree structures to specify the registry entries added or removed from the OS environment. The file section is a tree structure consisting of the files copied to C drive during the application installation. The profile section contains the initial set of block reference sequences during

Builder profiling of the application. The prefetch section consists of a subset of profiled blocks used by the Builder as a hint to the eStream client to prefetch initially. The comment section is used to inform the eStream client user of any relevant information about the application installation. Finally, the code section contains an optional program tailored for any application-specific installation not covered by the default eStream application installation procedure. In Windows version, the code section contains a Windows DLL.

Here is a detailed description of each fields of the AppInstallBlock.

Note: Little endian format is used for all the fields spanning more than 1 byte. Also, BlockNumber specifies blocks of 4K byte size.

# 1. Header Section:

The header section contains the basic information about that AppInstallBlock. This includes the versioning information, application identification,

**Core Header Structure:**

- **AibVersion [4 bytes]**: Magic number or appInstallBlock version number (which identifies the version of the appInstallBlock structure rather than the contents).
- **AppId [16 bytes]**: this is an application identifier unique for each application. On Windows, this identifier is the GUID generated from the 'guidgen' program. AppId for Word on Win98 will be different from Word on WinNT if it turns out that Word binaries are different between NT and 98.
- **VersionNo [4 bytes]**: Version number. This allows us to inform the client that the appInstallBlock has changed for a particular appId. This is useful for changes to the AppInstallBlock due to minor patch upgrades in the application.
- **ClientOSBitMap [4 bytes]**: Client OS supported bitmap or ID: for Win2K, Win98, WinNT and other future OSs we might support (it should be possible to say that this appInstallBlock is for more than one OS).
- **ClientOSServicePack [4 bytes]**: We might want to store the service pack level of the OS for which this appInstallBlock has been created. Note that when this field is set we cannot use multiple OS bits in the above field ClientOSBitMap.
- **Flags [4 bytes]**: Flags pertaining to AppInstallBlock
    - **Bit 0:** Reboot – If set, the eStream client needs to reboot the machine after installing the AppInstallBlock on the client machine.
    - **Bit 1:** Unicode – If set, the string characters are 2 bytes wide instead of 1 byte.
- **HeaderSize [2 bytes]**: Total size in bytes of the header section.
- **Reserved [32 bytes]**: Reserved spaces for future.

- **NumberOfSections [1 byte]**: Number of sections in the index table. This determines the number of entries in the index table structure described below:

**Index Table Structure: (variable number of entries)**

- **SectionType [1 bytes]**: The type of data describe in section. 0=file section, 1=variable section, 2=prefetch section, 3=profile section, 4=comment section, 5=code section.
- **SectionOffset [4 bytes]**: The offset from the beginning of the file indicates the beginning of section.
- **SectionSize [4 bytes]**: The size in bytes of section.

**Variable Structure:**

- **ApplicationNameLength [4 bytes]**: Byte size of the application name
- **ApplicationName [X bytes]**: Null terminating name of the application

## 2. File Section:

The file section contains a subset of the list of files needed by the application to run properly. This section does not enumerate files located in the standard application program directory. It consists of information about files copied into 'unusual' directory during the installation of an application. If the file content is small, the file is copied to the client machine. Otherwise, the file is relocated to the standard program directory suitable for streaming. The file section data is list of trees stored in a contiguous sequence of address space according to the pre-order traversal of the trees. A node in the tree can correspond to one or more levels of directory. A parent-child node pair is combined into a single node if the parent node has only a single child. Parsing the tree from the root of the tree to a leaf node results in a fully legal Windows pathname including the drive letter. Each entry of the node in the tree consists of the following structure:

**Directory Structure: (variable number of entries)**

- **Flags [4 byte]**: Bit 0 is set if this entry is a directory
- **NumberOfChildren [2 bytes]**: Number of nodes in this directory
- **DirectoryNameLength [4 bytes]**: Length of the directory name
- **DirectoryName [X bytes]**: Null terminating directory name

**Leaf Structure: (variable number of entries)**

- **Flags [4 byte]**: Bit 1 is set to 1 if this entry is a spoof or copied file name
- **FileVersion [4? bytes]**: Version of the file GetFileVersionInfo() if the file is win32 file image. Need variable file version size returned by GetFileVersionInfoSize(). Otherwise use GetFileTime() to retrieve the file creation time.
- **FileNameLength [4 bytes]**: Byte size of the file name

- **FileName [X bytes]**: Null terminating file name
- **DataLength [4 bytes]**: Byte size of the data. If spoof file, then data is the string of the spoof directory. If copied file, then data is the content of the file
- **Data [X bytes]**: Either the spoof file name or the content of the copied file

## 3. Add Variable and Remove Variable Sections:

The add and remove variable sections contain the system variable changes needed to run the application. In Windows system, each section consists of several number of registry subtrees. Each tree is stored in a contiguous sequence of address space according to the pre-order traversal of the tree. A node in the tree can correspond to one or more levels of directory in the registry. A parent-child node pair is combined into a single node if the parent node has only a single child. Parsing the tree from the root of the tree to a leaf node results in a fully legal key name. The order of the trees is shown here.

### a. Registry Subsection:

1. "HKCR": HKEY_CLASSES_ROOT
2. "HKCU": HKEY_CURRENT_USER
3. "HKLM": HKEY_LOCAL_MACHINE
4. "HKU": HKEY_USERS
5. "HKCC": HKEY_CURRENT_CONFIG

**Tree Structure: (5 entries)**

- **ExistFlag [1 byte]**: Set to 1 if this tree exist, 0 otherwise.
- **Key or Value Structure entries [X bytes]**: Serialization of the tree into variable number key or value structures described below.

**Key Structure: (variable number of entries)**

- **KeyFlag [1 byte]**: Set to 1 if this entry is a key or 0 if it's a value structure
- **NumberOfSubchild [4 bytes]**: Number of subkeys and values in this key directory
- **KeyNameLength [4 bytes]**: Byte size of the key name
- **KeyName [X bytes]**: Null terminating key name

**Value Structure: (variable number of entries)**

- **KeyFlag [1 byte]**: Set to 1 if this entry is a key or 0 if it's a value structure
- **ValueType [4 byte]**: Type of values from the Win32 API function RegQueryValueEx(): REG_SZ, REG_BINARY, REG_DWORD, REG_LINK, REG_NONE, etc...
- **ValueNameLength [4 bytes]**: Byte size of the value name
- **ValueName [X bytes]**: Null terminating value name

- **ValueDataLength [4 bytes]**: Byte size of the value data
- **ValueData [X bytes]**: Value of the Data

In addition to registry changes, an installation in Windows system may involve changes to the ini files. The following structure is used to communicate the ini file changes needed to be done on the eStream client machine. The ini entries are appended to the end of the variable section after the 5 registry trees are enumerated.

**b. INI Subsection: (not supported in eStream 1.0)**

- **NumFiles [4 bytes]**: Number of INI files modified.

**File Structure: (variable number of entries)**

- **FileNameLength [4 bytes]**: Byte length of the file name
- **FileName [X bytes]**: Name of the INI file
- **NumSection [4 bytes]**: Number of sections with the changes

**Section Structure: (variable number of entries)**

- **SectionNameLength [4 bytes]**: Byte length of the section name
- **SectionName [X bytes]**: Section name of an INI file
- **NumValues [4 bytes]**: Number of values in this section

**Value Structure: (variable number of entries)**

- **ValueLength [4 bytes]**: Byte length of the value data
- **ValueData [X bytes]**: Content of the value data

## 4. Prefetch Section:

The prefetch section contains a list of file blocks. The Builder profiler determines the set of file blocks critical for the initial run of the application. This data includes the code to start and terminate the application. It includes the file blocks containing for frequently used commands. For example, opening and saving of documents are frequently used commands and should be prefetched if possible. Another type of blocks to include in the prefetch section is the blocks associated with frequently accessed directories and file metadata in this directory. The prefetch section is divided into two subsections. One part contains the critical blocks that are used during startup of the streamed application. The second part consists of the blocks accessed for common user operations like opening and saving of document. The format of the data is described below:

**a. Critical Block Subsection:**

- **NumCriticalBlocks [4 bytes]:** Number of critical blocks.

**Block Structure: (variable number of entries)**

- **FileNumber [4 bytes]**: File Number of the file containing the block to prefetch
- **BlockNumber [4 bytes]**: Block Number of the file block to prefetch

**b. Common Block Subsection:**

- **NumCommonBlocks [4 bytes]:** Number of critical blocks.

**Block Structure: (variable number of entries)**

- **FileNumber [4 bytes]**: File Number of the file containing the block to prefetch
- **BlockNumber [4 bytes]**: Block Number of the file block to prefetch

## 5. Profile Section: (not used in eStream 1.0)

The profile section consists of a reference sequence of file blocks accessed by the application at runtime. Conceptually, the profile data is a two dimensional matrix. Each entry [*row, column*] of the matrix is the frequency a block *row* is followed by a block *column*. In any realistic applications of fair size, this matrix is very large and sparse. Proper data structure must be selected to store this sparse matrix efficiently in required storage space and minimize the overhead in accessing this data structure access.

The section is constructed from two basic structures: row and column structures. Each row structure is followed by N column structures specified in the NumberColumns field. Note that this is an optional section. But with appropriate profile data, the eStream client prefetcher performance can be increased.

**Row Structure: (variable number of entries)**

- **FileNumber [4 bytes]**: File Number of the row block
- **BlockNumber [4 bytes]**: Block Number of the row block
- **NumberColumns [4 bytes]**: number of blocks that follows this block. This field determines the number of column structures following this field.

**Column Structure: (variable number of entries)**

- **FileNumber [4 bytes]**: File Number of the column block
- **BlockNumber [4 bytes]**: Block Number of the column block
- **Frequency [4 bytes]**: frequency the row block is followed by column block

## 6. Comment Section:

The comment section is used by the Builder to describe this AppInstallBlock in more detail.

- **CommentLength [4 bytes]**: Byte size of the comment string
- **Comment [X bytes]**: Null terminating comment string

## 7. Code Section:

The code section consists of the application-specific initialization code needed to run on the eStream client to setup the client machine for this particular application. This section may be empty if the default initialization procedure in the eStream client is able to setup the client machine without requiring any application-specific instructions. On the Windows system, the code is a DLL file containing two exported function calls: *Install*(), *Uninstall*(). The eStream client loads the DLL and invokes the appropriate function calls.

- **CodeLength [4 bytes]**: Byte size of the code
- **Code [X bytes]**: Binary file containing the application-specific initialization code. On Windows, this is just a DLL file.

## 8. LicenseAgreement Section:

The Builder creates the license agreement section. The eStream client displays the license agreement text to the end-user before the application is started for the first time. The end-user must agree to all licensing agreement set by the software vendor in order to use the application.

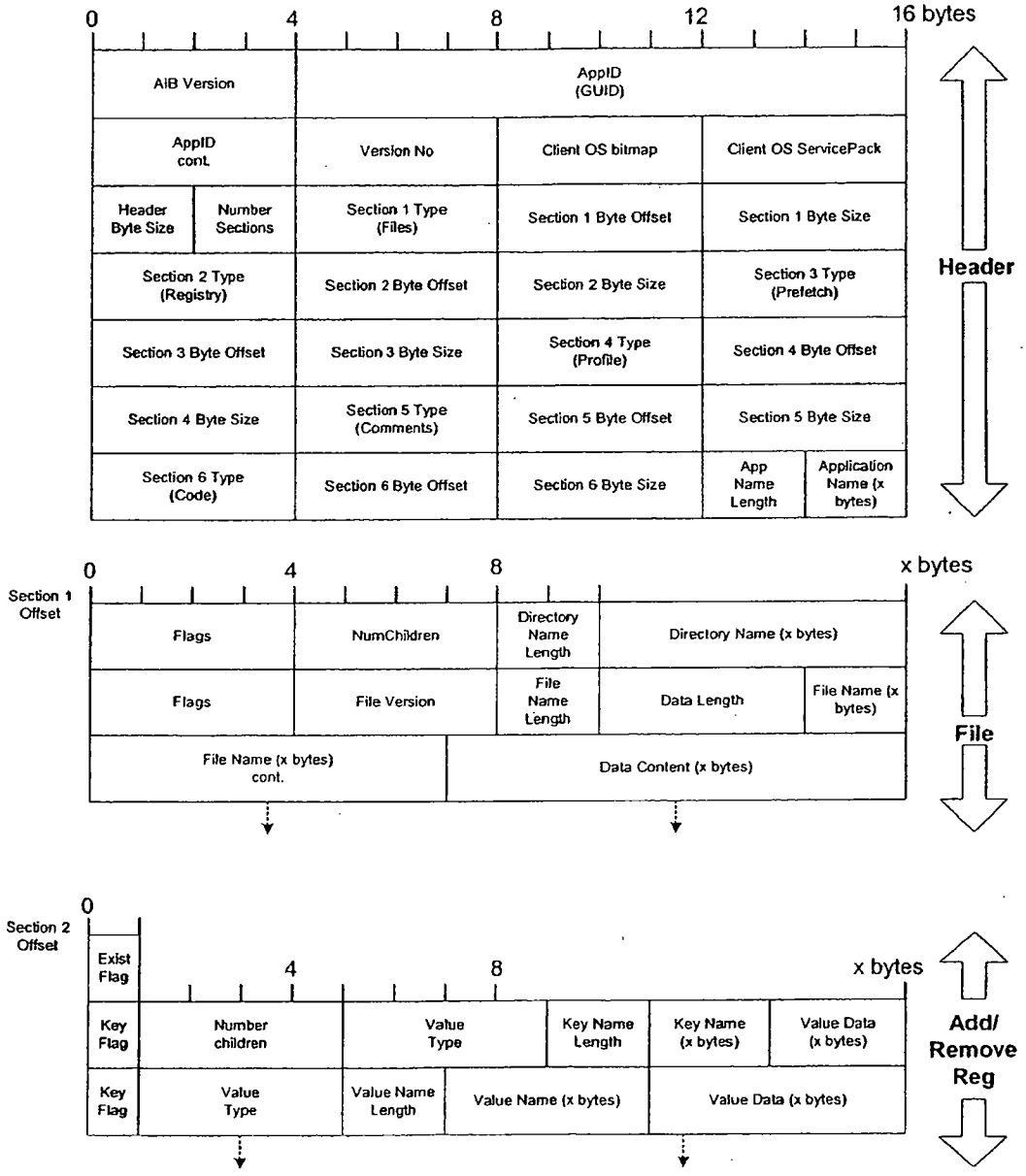
- **LicenseTextLength [4 bytes]**: Byte size of the license text
- **LicenseAgreement [X bytes]**: Null terminating license agreement string

# Open Issues

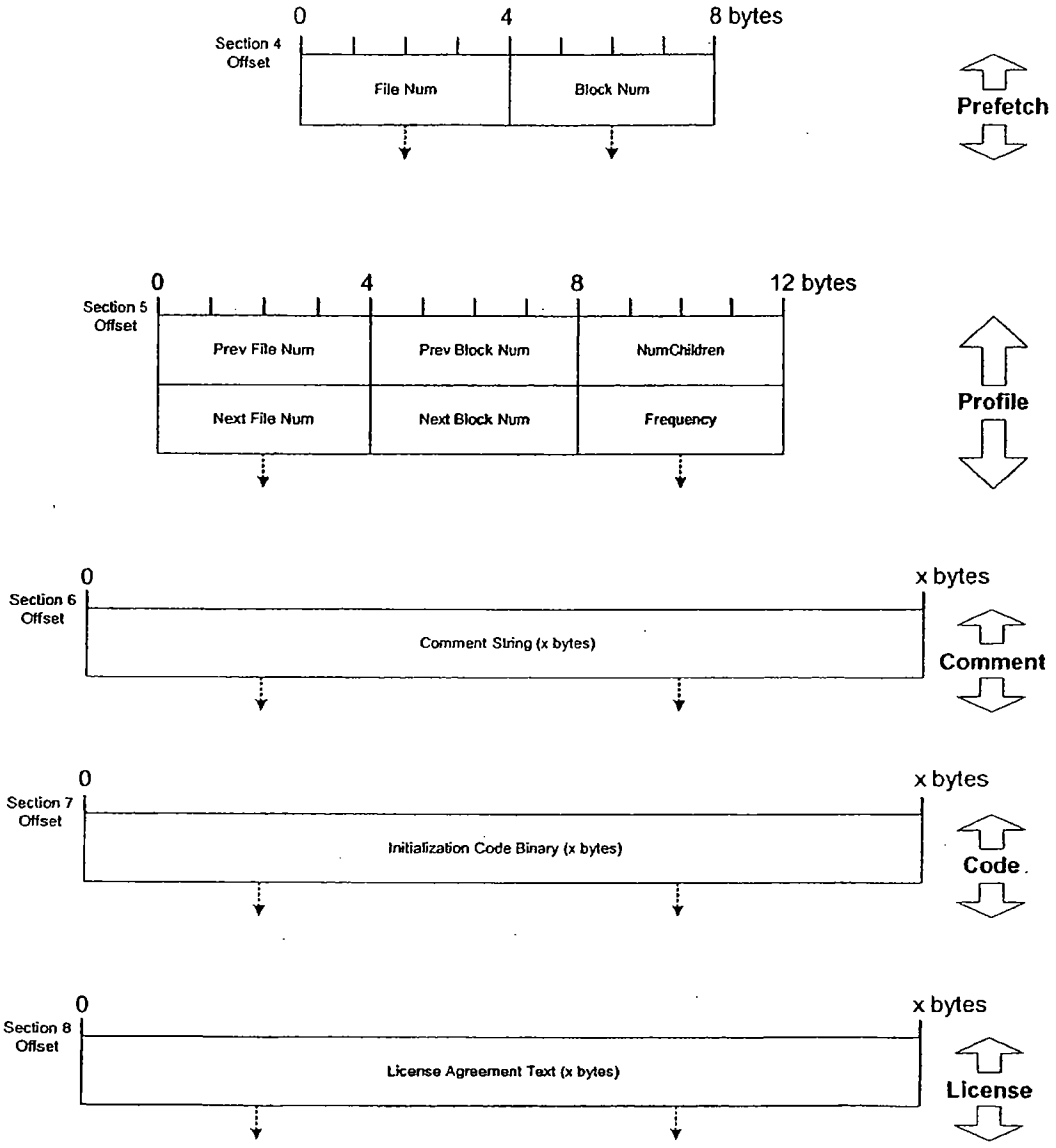
- What is the size of the AppInstallBlock for a typical application like Office?
- How large should the prefetch sections be for optimal run of an application? At minimum, it should contain at least start/termination code.
- How should the AppInstallBlock handle application license agreement text string? Add a new section or use comment section. Does the dialog need to have exactly the same interface as the license agreement dialog on the local installation?
- Currently, file section stores complete pathname including the drive letter. The installation may place files according to some variables like %System-

Root% or %UserProfile%. How does the Builder detect this so it can propagate this information to the client?

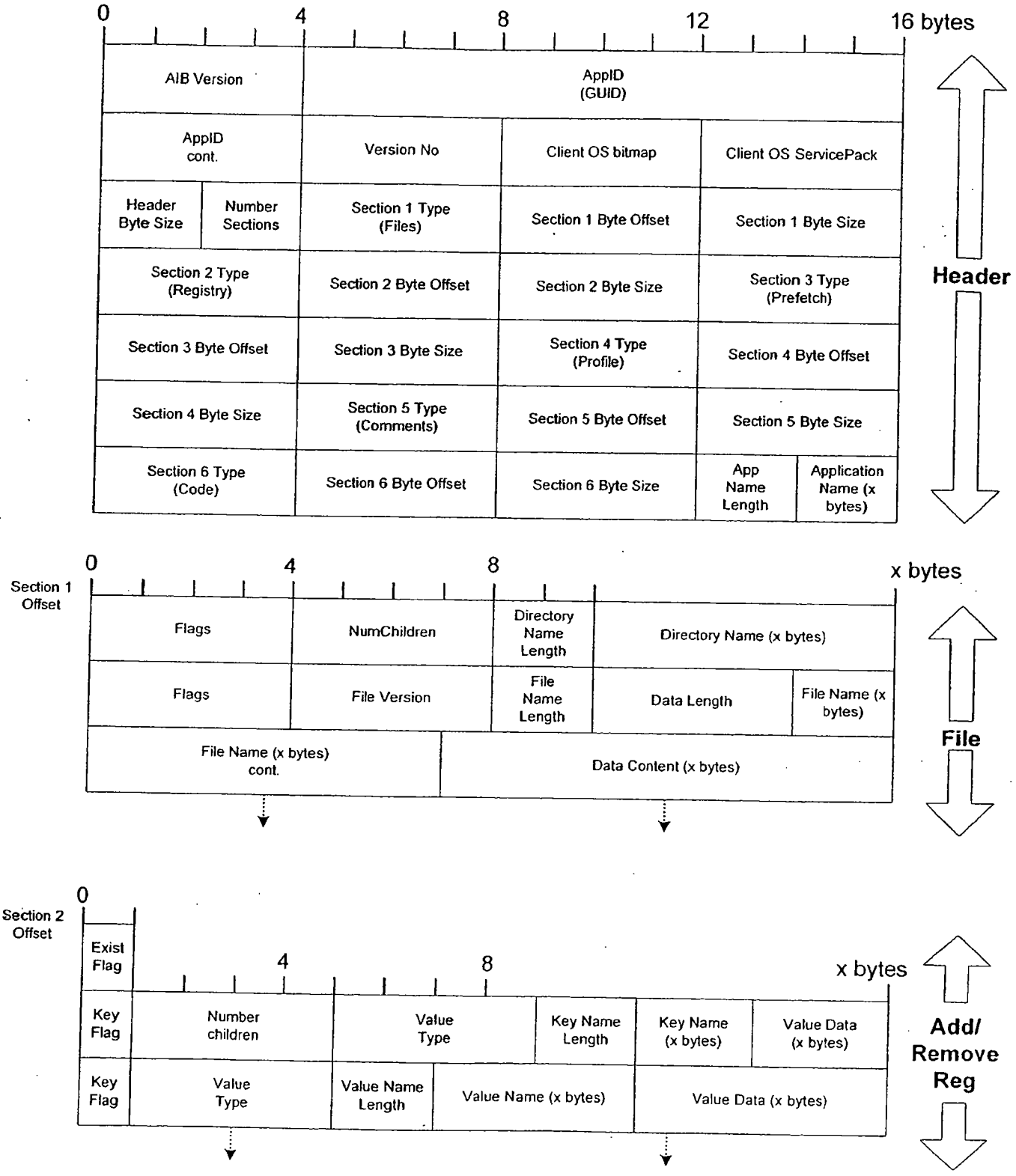
**Format of AppInstallBlock (part 1 of 2)**



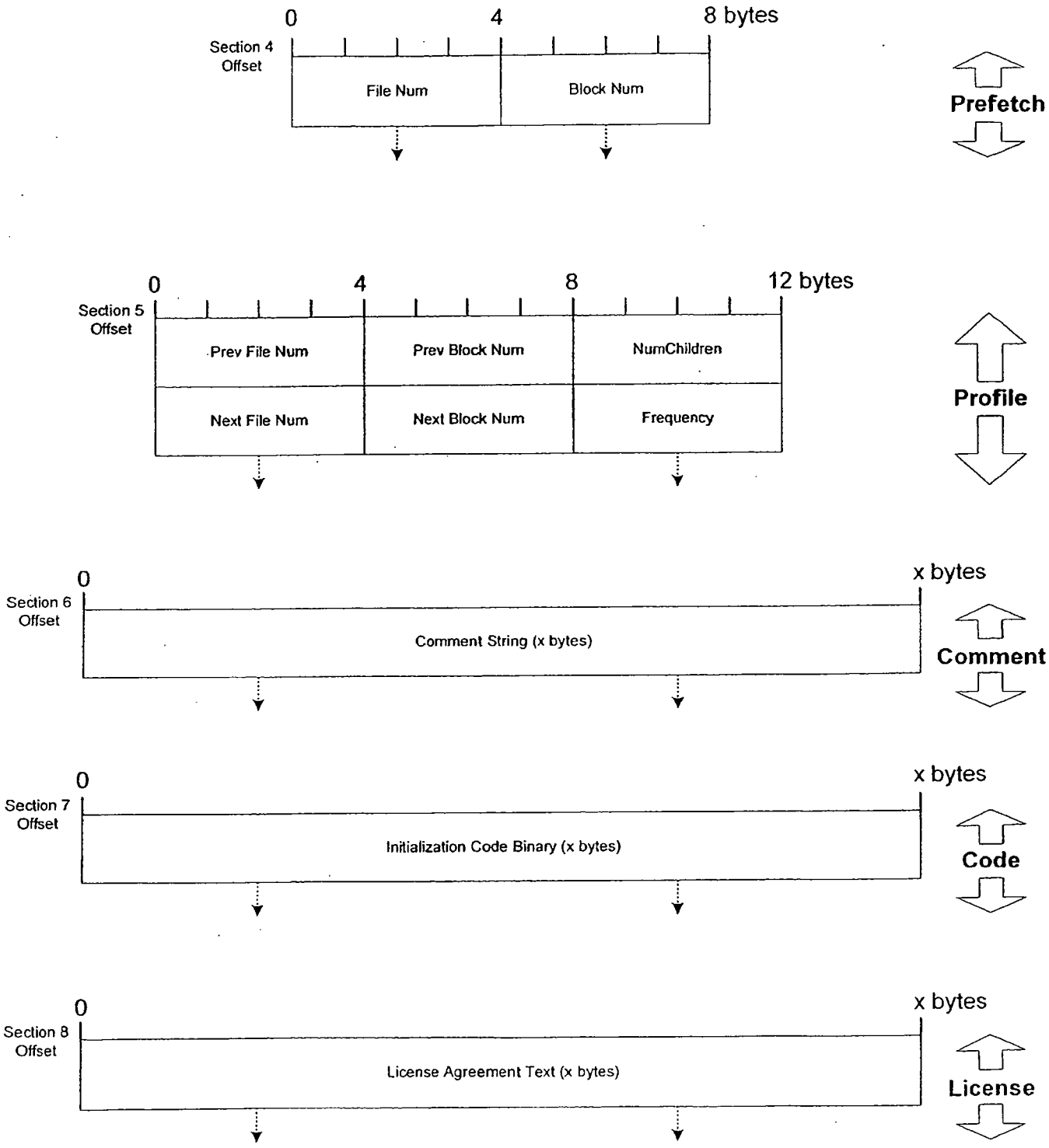
**Format of AppInstallBlock (part 2 of 2)**



# Format of AppInstallBlock (part 1 of 2)



# Format of AppInstallBlock (part 2 of 2)



# Converting Apps for Stream Delivery and subsequent Execution

The Streamed Application Set Builder is a software program. It is used to convert locally installable applications into a data set suitable for streaming over the network. The streaming-enabled data set is called the Streamed Application Set (SAS). This document describes the procedure used to convert locally installable applications into the SAS.

The application conversion procedure into the SAS consists of the several phases. In the first phase, the Builder program monitors the installation process of a local installation of the desired application for conversion. The Builder monitors any changes to the system and records those changes in an intermediate data structure. After the application is installed locally, the Builder enters the second phase of the conversion. In the second phase, the Builder program invokes the installed application executable and obtains sequences of frequently accessed file blocks of this application. Both the Builder program and the client software use the sequence data to optimize the performance of the streaming process. Once the sequencing information is obtained, the Builder enters the final phase of the conversion. In this phase, the Builder gathers all data obtained from the first two phase and processes the data into the Streamed Application Set.

In the next sections, detailed descriptions of the three phases of the Builder conversion process are described. The three phases consists of installation monitoring (IM), application profiling (AP), and finally SAS packaging (SP). In most cases, the conversion process is general and applicable to all type of system. In places where the conversion is OS dependent, the discussion is focused on Microsoft Windows environment. Issues on conversion procedure for other OS environment are described in later sections.

**Installation Monitoring (IM)**

In the first phase of the conversion process, the Builder Installation Monitor (IM) component invokes the application installation program that installs the application locally. The IM observes all changes to the local computer during the installation. The changes may involve one or more of the following: changes to system or environment variables; and modifications, addition, or deletion of one or more files. The IM records all changes to the variables and files in a data structure to be sent to the Builder's Streamed Application Packaging component. In the following paragraphs, detailed description of the Installation Monitor is described for Microsoft Windows environment.

In Microsoft Windows system, the Installation Monitor (IM) component consists of a kernel-mode driver subcomponent and a user-mode subcomponent. The kernel-mode driver is hooked into the system registry and file system function interface calls. The hook into the registry function calls allows the IM to monitor system variable changes. The hook into the file system function calls enables the IM to observe file changes.

**Installation Monitor Kernel-Mode subcomponent (IM-KM)**
The IM-KM subcomponent monitors two classes of information during an application installation: system registry modifications and file modifications. Different techniques are used for each of these classes.

To monitor system registry modifications, the IM-KM component replaces all kernel-mode API calls in the System Service Table that write to the system registry with new functions defined in the IM-KM subcomponent. When an installation program calls one of the API functions to write to the registry, the IM-KM function is called instead, which logs the modification data (including registry key path, value name and value data) and then forwards the call to the actual operating system defined function. The modification data is made available to the IM-UM subcomponent through a mechanism described below in **Installation Monitor User-Mode subcomponent (IM-UM).**

To monitor file modifications, a *filter driver* is attached to the file system's driver stack. Each time an installation program modifies a file on the system, a function is called in the IM-KM subcomponent, which logs the modification data (including file path and name) and makes it available to the IM-UM using a mechanism described below in **Installation Monitor User-Mode subcomponent (IM-UM).**

The mechanisms used for monitoring registry modifications and file modifications will capture modifications made by any of the processes currently active on the computer system. While the installation program is running, other processes that, for example, operate the desktop and service network connections may be running and may also modify files or registry data during the installation. This data must be removed from the modification data to avoid inclusion of modifications that are not part of the application installation. The IM-KM uses process monitoring to perform this filtering.

To do process monitoring, the IM-KM installs a process notification callback function that is called each time a process is created or destroyed by the operating system. Using this callback function, the operating system sends the **created** process ID as well as the process ID of the **creator** (or parent) process. The IM-KM uses this information, along with the process ID of the IM-UM, to create a list of all of the processes created during the application installation. The IM-KM uses the following algorithm to create this list:

1. Before the installation program is launched by the IM-UM, the IM-UM passes its own process ID to the IM-KM. Since the IM-UM is launching the installation application, the IM-UM will be the ancestor (parent, grandparent etc) of any process (with one exception – the Installer Service described below) that modifies files or registry data as part of the application installation.
2. When the installation is launched and begins the creating processes, the IM-KM process monitoring logic is notified by the operating system via the process notification callback function.
3. If the creator (parent) process ID sent to the process notification callback function is already in the process list, the new process is included in the list.

When an application on the system modifies either the registry or files, and the IM-KM monitoring logic captures the modification data, but before making it available to the IM-

UM, it first checks to see if the process that modified the registry or file is part of the process list. It is only made available to the IM-UM if it is in the process list.

It is possible that a process that is not a process ancestor of the IM-UM will make changes to the system as a proxy for the installation application. Using interprocess communication, an installation program may request than an *Installer Service* make changes to the machine. In order for the IM-KM to capture changes made by the Installer Service, the process monitoring logic includes a simple rule that also includes any registry or file changes that have been made by a process with the same name as the Installer Service process. On Windows 2000, for example, the Installer Service is called "msi.exe".

**Installation Monitor User-Mode subcomponent (IM-UM)**
The IM kernel-mode (IM-KM) driver subcomponent is controlled by the user-mode subcomponent (IM-UM). The IM-UM sends messages to the IM-KM to start and stop the monitoring process via standard I/O control messages known as IOCTLs. The message that starts the IM-KM also passes in the process ID of the IM-UM to facilitate process monitoring described in the IM-KM description.

When the installation program modifies the computer system, the IM-KM signals a named kernel event. The IM-UM listens for these events during the installation. When one of these events is signaled, the IM-UM calls the IM-KM using an IOCTL message. In response, the IM-KM packages data describing the modification and sends it to the IM-UM.

The IM-UM sorts this data and removes duplicates. Also, it parameterizes all local-system-specific registry keys, value names, and values. For example, an application will often store paths in the registry that allow it to find certain files at run-time. These path specification must be replaced with parameters that can be recognized by the client installation software.

A user interface is provided for the IM-UM that allows an operator of the Builder to browse through the changes made to the machine and to edit the modification data before the data is packaged into an SAS.

Once the installation of an application completed, the IM-UM forwards data structures representing the file and registry modifications to the Streamed Application Packager. See figure 1 for the control flow diagram of IM module.

**Monitoring Application Configuration**
Using the techniques described above for monitoring file modifications and monitoring registry modifications, the builder can also monitor a running application that is being configured for a particular working environment. The data acquired by the IM-UM can be used to duplicate the same configuration on multiple machines, making it unnecessary for each user to configure his/her own application installation.

An example of this is a client server application for which the client will be streamed to the client computer system. Common configuration modifications can be captured by the IM and packed into the SAS. When the application is streamed to the client machine, it is already configured to attach to the server and begin operation.

**Application Profiling (AP)**

In the second phase of the conversion process, the Builder's Application Profiler (AP) component invokes the application executable program that is installed during the first phase of the conversion process. Given a particular user input, the executable program file blocks are accessed in a particular sequence. And the purpose of the AP is to captures this sequence data associated with some user inputs. This data is useful in several ways.

First of all, frequently used file blocks can be streamed to the client machine before other less used file blocks. A frequently used file blocks is cached locally on the client cache before the user starts using the streamed application for the first time. This has the effect of making the streamed application as responsive to the user as the locally installed application by hiding any long network latency and bandwidth problems.

Secondly, the frequently accessed files can be reordered in the directory to allow faster lookup of the file information. This optimization is useful for directories with large number of files. When the client machine looks up a frequently used file in a directory, it finds this file early in the directory search. In an application run with many directory queries, the performance gain is significant.

Finally, the association of a set of file blocks with a particular user input allows the client machine to request minimum amount of data needed to respond to that particular user command. The profile data association with a user command is sent from the server to the client machine in the AppInstallBlock during the 'preparation' of the client machine for streaming. When the user on a client machine invokes a particular command, the codes corresponding to this command is prefetched from the server.

The Application Profiler (AP) is not as tied to the system as the Installation Monitor (IM) but there are still some OS dependent issues. In the Windows system, the AP still has two subcomponents: kernel-mode (AP-KM) subcomponent and the user-mode (AP-UM) subcomponent. The AP-UM invokes the converting application executable. Then AP-UM starts the AP-KM to track the sequences of file block accesses by the application. Finally when the application exits after the pre-specified amount of sequence data is gathered, the AP-UM retrieves the data from AP-KM and forwards the data to the Streamed Application Packager. See Figure 2 for the control flow diagram of the AP module.

**Streamed Application Set Packaging (SP)**

In the final phase of the conversion process, the Builder's Streamed Application Set Packager (SP) component processes the data structure from IM and AP to create a data set suitable for streaming over the network. This converted data set is called the

Streamed Application Set and is suitable for uploading to the Streamed Application Servers for subsequent downloading by the stream client. Figure 3 shows the control flow of the SP module.

Each file included in a Streamed Application Set is assigned a file number that identifies it within the SAS.

The Streamed Application Set consists of the three sets of data from the Streamed Application Server's perspective. The three types of data are the Concatenation Application File (CAF), the Size Offset File Table (SOFT), and the Root Versioning Table (RVT).

The Concatenation Application File (CAF) consists of all the files and directories needed to stream to the client. The CAF can be further divided into two subsets: initialization data set and the runtime data set.

The initialization data set is the first set of data to be streamed from the server to the client. This data set contains the information captured by IM and AP needed by the client to prepare the client machine for streaming this particular application. This initialization data set is also called the AppInstallBlock (AIB). In addition to the data captured by the IM and AP modules, the SP is also responsible for merging any new dynamic profile data gathered from the client and the server. This data is merged into the existing AppInstallBlock to optimize subsequent streaming of the application. With the list of files obtained by the IM during application installation, the SP module separates the list of files into regular streamed files and the spoof files. The spoof files consists of those files not installed into standard application directory. This includes files installed into system directories and user specific directories. The detailed format description of the AppInstallBlock is described in Appendix B.

The second part of the CAF consists of the runtime data set. This is the rest of the data that is streamed to the client once the client machine is initialized for this particular application. The runtime data consists of all the regular application files and the directories containing information about those application files. Detailed format description of the runtime data in the CAF section is described in Appendix A. The SP appends every files recorded by IM into the CAF and generates all directories. Each directory contains list of file name, file number, and the metadata associated with the files in that particular directory.

The SP is also responsible for generating the SOFT file. This is a table used to index into the CAF for determining the start and the end of a file. The server uses this information to quickly access the proper file within the directory for serving the proper file blocks to the client.

Finally, the SP creates the RVT file. The Root Versioning Table contains a list of root file number and version number. This information is used to track minor application patches and upgrades. Each entry in the RVT corresponds to one patch level of the

application with a corresponding new root directory. The SP generates new parent directories when any single file in that subdirectory tree is changed from the patched upgrade. The RVT is uploaded to the server and requested by the client at appropriate time for the most updated version of the application by a simple comparison of the client's Streamed Application root file number with the RVT table located on the server once the client is granted access authorization to retrieve the data. Figure 4 shows the internal representation of a simple SAS before and after a new file is added to a new version of an application.

**Data Flow Description**

The following list describes the data that is passed from one component to another. The numbers corresponds to the numbering in the Data Flow diagram.

**Install Monitor**

1. The full pathname of the installer program is query from the user of the Builder program and is sent to the Install Monitor.
2. The Install Monitor (IM) user-mode sends a read request to the OS to spawn a new process for installing the application on the local machine.
3. The OS loads the application installer program into memory and run the application installer program. OS returns the process ID to the IM.
4. The application program is started by the IM-UM.
5. The application installer program sends read request to the OS to read the content of the CD.
6. The CD media data files are read from the CD.
7. The files are written to the appropriate location on the local hard-drive.
8. IM kernel-mode captures all file read/write requests and all registry read/write requests by the application installer program.
9. IM user-mode program starts the IM kernel-mode program and sends the request to start capturing all relevant file and registry data.
10. IM kernel-mode program sends the list of all file modifications, additions, and deletions; and all registry modifications, additions, and deletions to the IM user-mode program.
11. Inform the SAS Builder UI that the installation monitoring has completed and display the file and registry data in a graphical user interface.

**Application Profiler**

12. Builder UI invokes Application Profiling (AP) user-mode program by querying the user for the list of application executable names to be profiled. The AP user-mode also query the user for division of file blocks into sections corresponding to the commands invoked by the user of the application being profiled.
13. Application Profiler user-mode invokes each application executable in succession by spawning each program in a new process. The OS loads the application executable into memory, run the application executable, and return the process ID to the Application Profiler.

14. During execution, the OS on behalf of the application send the request to the hard-drive controller to read the appropriate file blocks into memory as needed by the application.
15. The hard-drive controller returns all file blocks requested by the OS.
16. Every file accesses to load the application file blocks into memory are monitored by the Application Profiler (AP) kernel-mode program.
17. The AP user-mode program informs the AP kernel-mode program to start monitoring relevant file accesses.
18. Application Profiler kernel-mode returns the file access sequence and frequency information to the user-mode program.
19. Application Profiler returns the processed profile information. This has two sections. The first section is used to identify frequency of files accessed. The second section is used to list the file blocks for prefetch to the client. The file blocks can be further categorized into subsections according to the commands invoked by the user of the application.

**SAS Packager**

20. The Streamed Application Packager receives files and registry changes from the Builder UI. It also receives the file access frequency and a list of file blocks from the Profiler. File numbers are assigned to each file.
21. The Streamed Application Packager reads all file data from the hard-drive that are copied there by the application installer.
22. The Streamed Application Packager also reads the previous version of Streamed Application Set for support of minor patch upgrades.
23. Finally, the new Streamed Application Set data is stored back to non-volatile storage.
24. For new profile data gathered after the SAS has been created, the packager is invoked to update the AppInstallBlock in the SAS with the new profile information.

**Mapping of Data Flow to Streamed Application Set (SAS)**

Step 7: Data gathered from this step consist of the registry and file modification, addition, and deletion. This data is mapped to the AppInstallBlock's File Section, Add Registry Section, and Remove Registry Section.

Step 8 & 19: File data are copied to the local hard-drive then concatenated into part of the CAF contents. Part of the data is identified as spoof or copied files and the file names and/or contents are added to the AppInstallBlock.

Step 15 & 21: Part of the data gathered by the Profiler or gathered dynamically by the client is used in the AppInstallBlock as a prefetch hint to the client. Another part of the data is used to generate a more efficient SAS Directory content by ordering the files according the usage frequency.

Step 20: If the installation program was an upgrade, SAS Packager needs previous version of the Streamed Application Set data. Appropriate data from the previous version is combined with the new data to form the new Streamed Application Set.

**Format of Streamed Application Set**

The format of the Streamed Application Set consists of 3 sections: Root Version Table (RVT), Size Offset File Table (SOFT), and Concatenation Application File (CAF). The RVT section lists all versions of the root file numbers available in a Streamed Application Set. The SOFT section consists of the pointers into the CAF section for every file in the CAF. The CAF section contains the concatenation of all the files associated with the streamed application. The CAF section is made up of regular application files, SAS directory files, AppInstallBlock, and icon files. Please see Appendix A for detailed information on the content of the SAS file. Figure 6 shows a typical layout of the SAS file.

**OS dependent format**

The format of the Streamed Application Set is designed to be as portable as possible across all OS platforms. At the highest level, the format of CAF, SOFT, and RVT that make up the format of Streamed Application Set are completely portable across any OS platforms. One piece of data structure that is OS dependent is located in the initialization data set called AppInstallBlock in the CAF. This data is dependent on the type of OS due to the differences in low-level system differences among different OS. For example, the Microsoft Windows contain system environment variables called the Registry. The Registry has a particular tree format not found in other operating systems like UNIX or MacOS.

Another OS dependent piece of data is located in the SAS directory files in the CAF. The directory contains file metadata information specific to Windows files. For example on the UNIX platform, there does not exist a hidden flag. This platform specific information needs to be transmitted to the client to fool the streamed application into believing that the application data is located natively on the client machine with all the associated file metadata in tack. If SAS is to be used to support streaming of UNIX or MacOS applications, file metadata specific to those systems will need to be recorded in the SAS directory.

Lastly, the format of the file names itself is OS dependent. Applications running on the Windows environment inherit the old MSDOS 8.3 file name format. To support this properly, the format of the SAS Directory file in CAF requires an additional 8.3 field to store this information. This field is not needed in other operating systems like UNIX or MacOS.

**Device driver versus file system paradigm**

The SAS client Prototype is implemented using the 'device driver' paradigm. One of the advantages of the device driver approach is that the caching of the sector blocks is simpler. In the device driver approach, the client cache manager only needs to track sector number in its cache. In comparison with the 'file system' paradigm, more complex data structure is required to track a subset of a file that is cached on a client machine. This makes 'device driver' paradigm easier to implement.

On the other hand, there are many drawbacks to the 'device driver' paradigm. On the Windows system, the device driver approach has problem supporting large number of

applications. This is due to the limitation on the number of assignable drive letters available in a Windows system (26 letters); and the fact that each application needs to be located on its own device. Note that having multiple applications on a device is possible, but then the server needs to maintain exponential number of devices that support all possible combinations of applications. This is too costly to maintain on the server.

Another problem with the device driver approach is that the device driver operates at the disk sector level. This is a much lower level than operating at the file level in the file system approach. The device driver does not know anything about files. Thus, the device driver cannot easily interact with the file level issues. For example, spoofing files and interacting with OS buffer cache is nearly impossible with device driver approach. But both spoofing files and interacting with OS buffer cache is need to get higher performance.

See figure 7 and 8 for the differences between two paradigms.

**Implementation in the Prototype**

The prototype has been implemented and tested successfully on the Windows and Linux distributed system. The prototype is implemented using the 'device driver' paradigm as described above. The exact procedure for streaming application data is described next.

First of all, the prototype server is started on either the Windows-based or Linux-based system. The server creates a large local file mimicking large local disk images. Once the disk images are prepared, it listens to TCP/IP ports for any disk sector read or write requests.

Secondly, the conversion process is done on a Windows system via semi-manual procedure. The server disk image is 'mounted' on the local Z drive by making the proper TCP/IP connection to the server. Then the application installation program is invoked and the application is installed into the Z drive. This writes the application files into the Z drive device driver, through the TCP/IP connection, and finally on to the server disk image. At the same time, a file and registry monitoring program records all registry and file changes. This data is stored as an initialization file to be invoked on the client to prepare the client machine for streaming.

Finally, after the application files is stored on the server disk image, the client prototype is started. The client connects to the server and 'mount' the server disk image as a local Z drive. Then the initialization file is invoked which setup the local registry variables and copy system files into proper directories. Once the local machine is prepared for streaming that particular application, the user can start using the application. When the application is first started, the pages are not located in the local buffer cache. The OS makes sector request to the Streamed Application device driver that forwards the sector request to the Streamed Application Cache Manager. If the sector is located in the Streamed Application cache, then the data is returned immediately. If the data is not located in the Streamed Application cache, then the request forwarded to the network component that sends the message to the server. The server finds the proper sector data

and returns the data to the client. The client Streamed Application Cache Manager caches the new sector data and forwards the sector data to the Streamed Application device driver. The device driver returns the sector data to the OS.

**Implementation of SAS Builder**

The SAS Builder has been implemented on the Windows-based platform. A preliminary Streamed Application Set file has been created for real-world applications like Adobe Photoshop. A simple extractor program has been developed to extract the SAS data on a pristine machine without the application installed locally. Once the extractor program is run on the SAS, the application runs as if it was installed locally on that machine. This process verifies the correctness of the SAS Building process.

# Appendix A: Format of Streamed Application Set (SAS)

## Functionality

The streamed application set is a data set associated with an application suitable for streaming over the network. The SAS is generated by the SAS Builder program. This program converts locally installable applications into the SAS. This document describes the format of the SAS.

**Note:** Fields greater than a single byte is stored in little-endian format. The Stream Application Set (SAS) file size is limited to 2^64 bytes. The files in the CAF section are laid out in the same order as its corresponding entries in the SOFT table.

## Data type definitions

The format of the SAS consists of 4 sections: header, Root Version Table (RVT), Size Offset File Table (SOFT), and Concatenation Application File (CAF) sections.

### 1. Header section

- **MagicNumber [4 bytes]:** Magic number identifying the file content with the SAS.
- **ESSVersion [4 bytes]:** Version number of the SAS file format.
- **AppID [16 bytes]:** A unique application ID for this application. This field must match the AppID located in the AppInstallBlock. Window Guidgen API is used to create this identifier.
- **Flags [4 bytes]:** Flags pertaining to SAS.
- **Reserved [32 bytes]:** Reserved spaces for future.

- **RVToffset [8 bytes]:** Byte offset into the start of the RVT section.
- **RVTsize [8 bytes]:** Byte size of the RVT section.
- **SOFToffset [8 bytes]:** Byte offset into the start of the SOFT section.
- **SOFTsize [8 bytes]:** Byte size of the SOFT section.
- **CAFoffset [8 bytes]:** Byte offset into the start of the CAF section.
- **CAFsize [8 bytes]:** Byte size of the CAF section.

- **VendorNameIsAnsi [1 byte]:** 0 if the vendor name is in Unicode format. 1 if the vendor name is in ANSI format.
- **VendorNameLength [4 bytes]:** Byte length of the vendor name.
- **VendorName [X bytes]:** Name of the software vendor who created this application. I.e. "Microsoft". Null-terminated.
- **AppBaseNameIsAnsi [1 byte]:** 0 if the vendor name is in Unicode format. 1 if the vendor name is in ANSI format.
- **AppBaseNameLength [4 bytes]:** Byte length of the application base name.

- **AppBaseName [X bytes]:** Base name of the application. I.e. "Word 2000". Null-terminated.
- **MessageIsAnsi [1 byte]:** 0 if the vendor name is in Unicode format. 1 if the vendor name is in ANSI format.
- **MessageLength [4 bytes]:** Byte length of the message text.
- **Message [X bytes]:** Message text. Null-terminated.

## 2. Root Version Table (RVT) section

The Root version entries are ordered in a decreasing value according to their file numbers. The Builder generates unique file numbers within each SAS in a monotonically increasing value. So larger root file number implies later versions of the same application. The latest root version is located at the top of the section to allow the SAS Server easy access to the data associated with the latest root version.

- **NumberEntries [4 bytes]:** Number of patch versions contained in this SAS. The number indicates the number of entries in the Root Version Table (RVT).
-

**Root Version structure: (variable number of entries)**

- **VersionNumber [4 bytes]:** Version number of the root directory.
- **FileNumber [4 bytes]:** File number of the root directory.
- **VersionNameIsAnsi [1 byte]:** 0 if the vendor name is in Unicode format. 1 if the vendor name is in ANSI format.
- **VersionNameLength [4 bytes]:** Byte length of the version name
- **VersionName [X bytes]:** Application version name. I.e. "SP 1".
- **Metadata [32 bytes]:** See SAS FS Directory for format of the metadata.

## 3. Size Offset File Table (SOFT) section

The SOFT table contains information to locate specific files in the CAF section. The entries are ordered according to the file number starting from 0 to NumberFiles-1. The start of the SOFT table is aligned to 8 bytes boundary for faster access.

**SOFT entry structure: (variable number of entries)**

- **Offset [8 bytes]:** Byte offset into CAF of the start of this file.
- **Size [8 bytes]:** Byte size of this file. The file is located from address Offset to Offset+Size.

## 4. Concatenation Application File (CAF) section

CAF is a concatenation of all file or directory data into a single data structure. Each piece of data can be a regular file, an AppInstallBlock, an SAS FS directory file, or an icon file.

### a. *Regular Files*

- **FileData [X bytes]:** Content of a regular file

### b. *AppInstallBlock (See AppInstallBlock document for detail format)*

A simplified description of the AppInstallBlock is listed here. For exact detail of the individual fields in the AppInstallBlock, please see Appendix B.

- **Header section [X bytes]:** Header for AppInstallBlock containing information to identify this AppInstallBlock.
- **Files section [X bytes]:** Section containing file to be copied or spoofed.
- **AddVariable section [X bytes]:** Section containing system variables to be added.
- **RemoveVariable section [X bytes]:** Section containing system variables to be removed.
- **Prefetch section [X bytes]:** Section containing pointers to files to be prefetched to the client.
- **Profile section [X bytes]:** Section containing profile data.
- **Comment section [X bytes]:** Section containing comments about AppInstallBlock.
- **Code section [X bytes]:** Section containing application-specific code needed to prepare local machine for streaming this application
- **LicenseAgreement section [X bytes]:** Section containing licensing agreement message.

### c. *SAS Directory*

An SAS Directory contains information about the subdirectories and files located within this directory. This information is used to store metadata information related to the files associated with the streamed application. This data is used to fool application into thinking that it is running locally on a machine when most of the data is resided elsewhere.

The SAS directory contains information about files in its directory. The information includes file number, names, and metadata associated with the files.

- **MagicNumber [4 bytes]:** Magic number for SAS directory file.
- **ParentFileID [16+4 bytes]:** AppID+FileNumber of the parent directory. AppID is set to 0 if the directory is the root.
- **SelfFileID [16+4 bytes]:** AppID+FileNumber of this directory.
- **NumFiles [4 bytes]:** Number of files in the directory.

**Variable-Sized File Entry:**

- **UsedFlag [1 byte]:** 1 for used, 0 for unused.
- **ShortLen [1 byte]:** Length of short file name.
- **LongLen [2 byte]:** Length of long file name.
- **NameHash [4 bytes]:** Hash value of the short file name for quick lookup without comparing whole string.
- **ShortName [24 bytes]:** 8.3 short file name in Unicode. Not null-terminated.
- **FileID [16+4 bytes]:** AppID+FileNumber of each file in this directory.
- **Metadata [32 bytes]:** The metadata consists of file **byte size** (8 bytes), file **creation time** (8 bytes), file **modified time** (8 bytes), **attribute flags** (4 bytes), **SAS flags** (4 bytes). The bits of the **attribute flags** have the following meaning:
  - **Bit 0:** Read-only – Set if file is read-only
  - **Bit 1:** Hidden – Set if file is hidden from user
  - **Bit 2:** Directory – Set if the file is an SAS Directory
  - **Bit 3:** Archive – Set if the file is an archive
  - **Bit 4:** Normal – Set if the file is normal
  - **Bit 5:** System – Set if the file is a system file
  - **Bit 6:** Temporary – Set if the file is temporary

  The bits of the **SAS flags** have the following meaning:
  - **Bit 0:** ForceUpgrade – Used only on root file. Set if client is forced to upgrade to this particular version if the current root version on the client is older.
  - **Bit 1:** RequireAccessToken – Set if file require access token before client can read it.
  - **Bit 2:** Read-only – Set if the file is read-only
- **LongName [X bytes]:** Long filename in Unicode format with null-termination character.

### *d. Icon files*

- **IconFileData [X bytes]:** Content of an icon file.

# Appendix B: Format of AppInstallBlock

## Functionality

The AppInstallBlock is a block of code and data associated with a particular application. This AppInstallBlock contains the information needed to by the SAS client to 'initialize' the client machine before the streamed application is used for the first time. It also contains optional profiling data for increasing the runtime performance of that streamed application.

The AppInstallBlock is created offline by the SAS Builder program. First of all, the Builder monitors the installation process of a local version of the application installation program and records changes to the system. This includes any environment variables added or removed from the system, and any files added or modified in the system directories. Files added to the application specific directory is not recorded in the AppInstallBlock to reduce the amount of time needed to send the AppInstallBlock to the SAS client. Secondly, the Builder profiles the application to obtain the list of critical pages needed to run the application initially and an initial page reference sequence of the pages accessed during a sample run of the application. The AppInstallBlock contains an optional application-specific initialization code. This code is needed when the default initialization procedure is insufficient to setup the local machine environment for that particular application.

The AppInstallBlock and the runtime data are packaged into the SAS by the Builder and then uploaded to the application server. After the SAS client subscribed to an application and before the application is run for the first time, the AppInstallBlock is send by the server to the client. The SAS client invokes the default initialization procedure and the optional application-specific initialization code. Together, the default and the application-specific initialization procedure process the data in the AppInstallBlock to make the machine ready for streaming that particular application.

## Data type definitions

The AppInstallBlock is divided into the following sections: header section, variable section, file section, profile section, prefetch section, comment section, and code section. The header section contains general information about the AppInstallBlock. The information includes the total byte size and an index table containing size and offset into other sections. In Windows version, the variable section consists of two registry tree structures to specify the registry entries added or removed from the OS environment. The file section is a tree structure consisting of the files copied to C drive during the application installation. The profile section contains the initial set of block reference sequences during Builder profiling of the application. The prefetch section consists of a subset of profiled blocks used by the Builder as a hint to the SAS client to prefetch initially. The comment section is used to inform the SAS client user of any relevant

information about the application installation. Finally, the code section contains an optional program tailored for any application-specific installation not covered by the default streamed application installation procedure. In Windows version, the code section contains a Windows DLL. The following is a detailed description of each fields of the AppInstallBlock.

Note: Little endian format is used for all the fields spanning more than 1 byte. Also, BlockNumber specifies blocks of 4K byte size.

# 1. Header Section:

The header section contains the basic information about this AppInstallBlock. This includes the versioning information, application identification, and index into other sections of the file.

**Core Header Structure:**

- **AibVersion [4 bytes]**: Magic number or appInstallBlock version number (which identifies the version of the appInstallBlock structure rather than the contents).
- **AppId [16 bytes]**: this is an application identifier unique for each application. On Windows, this identifier is the GUID generated from the 'guidgen' program. AppId for Word on Win98 will be different from Word on WinNT if it turns out that Word binaries are different between NT and 98.
- **VersionNo [4 bytes]**: Version number. This allows us to inform the client that the appInstallBlock has changed for a particular appId. This is useful for changes to the AppInstallBlock due to minor patch upgrades in the application.
- **ClientOSBitMap [4 bytes]**: Client OS supported bitmap or ID: for Win2K, Win98, WinNT and other future OSs we might support (it should be possible to say that this appInstallBlock is for more than one OS).
- **ClientOSServicePack [4 bytes]**: We might want to store the service pack level of the OS for which this appInstallBlock has been created. Note that when this field is set we cannot use multiple OS bits in the above field ClientOSBitMap.
- **Flags [4 bytes]**: Flags pertaining to AppInstallBlock
  - **Bit 0:** Reboot – If set, the SAS client needs to reboot the machine after installing the AppInstallBlock on the client machine.
  - **Bit 1:** Unicode – If set, the string characters are 2 bytes wide instead of 1 byte.
- **HeaderSize [2 bytes]**: Total size in bytes of the header section.
- **Reserved [32 bytes]**: Reserved spaces for future.
- **NumberOfSections [1 byte]**: Number of sections in the index table. This determines the number of entries in the index table structure described below:

**Index Table Structure: (variable number of entries)**

- **SectionType [1 bytes]**: The type of data describe in section. 0=file section, 1=variable section, 2=prefetch section, 3=profile section, 4=comment section, 5=code section.
- **SectionOffset [4 bytes]**: The offset from the beginning of the file indicates the beginning of section.
- **SectionSize [4 bytes]**: The size in bytes of section.

**Variable Structure:**

- **ApplicationNameIsAnsi [1 byte]**: 1 if ansi, 0 if Unicode.
- **ApplicationNameLength [4 bytes]**: Byte size of the application name
- **ApplicationName [X bytes]**: Null terminating name of the application

## 2. File Section:

The file section contains a subset of the list of files needed by the application to run properly. This section does not enumerate files located in the standard application program directory. It consists of information about files copied into 'unusual' directory during the installation of an application. If the file content is small, the file is copied to the client machine. Otherwise, the file is relocated to the standard program directory suitable for streaming. The file section data is list of trees stored in a contiguous sequence of address space according to the pre-order traversal of the trees. A node in the tree can correspond to one or more levels of directory. A parent-child node pair is combined into a single node if the parent node has only a single child. Parsing the tree from the root of the tree to a leaf node results in a fully legal Windows pathname including the drive letter. Each entry of the node in the tree consists of the following structure:

**Directory Structure: (variable number of entries)**

- **Flags [4 byte]**: Bit 0 is set if this entry is a directory
- **NumberOfChildren [2 bytes]**: Number of nodes in this directory
- **DirectoryNameLength [4 bytes]**: Length of the directory name
- **DirectoryName [X bytes]**: Null terminating directory name

**Leaf Structure: (variable number of entries)**

- **Flags [4 byte]**: Bit 1 is set to 1 if this entry is a spoof or copied file name
- **FileVersion [8 bytes]**: Version of the file GetFileVersionInfo() if the file is win32 file image. Need variable file version size returned by GetFileVersionInfoSize(). Otherwise use file size or file modified time to compare which file is the later version.
- **FileNameLength [4 bytes]**: Byte size of the file name
- **FileName [X bytes]**: Null terminating file name

- **DataLength [4 bytes]**: Byte size of the data. If spoof file, then data is the string of the spoof directory. If copied file, then data is the content of the file
- **Data [X bytes]**: Either the spoof file name or the content of the copied file

## 3. Add Variable and Remove Variable Sections:

The add and remove variable sections contain the system variable changes needed to run the application. In Windows system, each section consists of several number of registry subtrees. Each tree is stored in a contiguous sequence of address space according to the pre-order traversal of the tree. A node in the tree can correspond to one or more levels of directory in the registry. A parent-child node pair is combined into a single node if the parent node has only a single child. Parsing the tree from the root of the tree to a leaf node results in a fully legal key name. The order of the trees is shown here.

### a. Registry Subsection:

1. "HKCR": HKEY_CLASSES_ROOT
2. "HKCU": HKEY_CURRENT_USER
3. "HKLM": HKEY_LOCAL_MACHINE
4. "HKUS": HKEY_USERS
5. "HKCC": HKEY_CURRENT_CONFIG

**Tree Structure: (5 entries)**

- **ExistFlag [1 byte]**: Set to 1 if this tree exist, 0 otherwise.
- **Key or Value Structure entries [X bytes]**: Serialization of the tree into variable number key or value structures described below.

**Key Structure: (variable number of entries)**

- **KeyFlag [1 byte]**: Set to 1 if this entry is a key or 0 if it's a value structure
- **NumberOfSubchild [4 bytes]**: Number of subkeys and values in this key directory
- **KeyNameLength [4 bytes]**: Byte size of the key name
- **KeyName [X bytes]**: Null terminating key name

**Value Structure: (variable number of entries)**

- **KeyFlag [1 byte]**: Set to 1 if this entry is a key or 0 if it's a value structure
- **ValueType [4 byte]**: Type of values from the Win32 API function RegQueryValueEx(): REG_SZ, REG_BINARY, REG_DWORD, REG_LINK, REG_NONE, etc...
- **ValueNameLength [4 bytes]**: Byte size of the value name
- **ValueName [X bytes]**: Null terminating value name
- **ValueDataLength [4 bytes]**: Byte size of the value data

- **ValueData [X bytes]**: Value of the Data

In addition to registry changes, an installation in Windows system may involve changes to the ini files. The following structure is used to communicate the ini file changes needed to be done on the SAS client machine. The ini entries are appended to the end of the variable section after the 5 registry trees are enumerated.

**b. INI Subsection:**

- **NumFiles [4 bytes]**: Number of INI files modified.

**File Structure: (variable number of entries)**

- **FileNameLength [4 bytes]**: Byte length of the file name
- **FileName [X bytes]**: Name of the INI file
- **NumSection [4 bytes]**: Number of sections with the changes

**Section Structure: (variable number of entries)**

- **SectionNameLength [4 bytes]**: Byte length of the section name
- **SectionName [X bytes]**: Section name of an INI file
- **NumValues [4 bytes]**: Number of values in this section

**Value Structure: (variable number of entries)**

- **ValueLength [4 bytes]**: Byte length of the value data
- **ValueData [X bytes]**: Content of the value data

## 4. Prefetch Section:

The prefetch section contains a list of file blocks. The Builder profiler determines the set of file blocks critical for the initial run of the application. This data includes the code to start and terminate the application. It includes the file blocks containing for frequently used commands. For example, opening and saving of documents are frequently used commands and should be prefetched if possible. Another type of blocks to include in the prefetch section is the blocks associated with frequently accessed directories and file metadata in this directory. The prefetch section is divided into two subsections. One part contains the critical blocks that are used during startup of the streamed application. The second part consists of the blocks accessed for common user operations like opening and saving of document. The format of the data is described below:

**a. Critical Block Subsection:**

- **NumCriticalBlocks [4 bytes]:** Number of critical blocks.

**Block Structure: (variable number of entries)**

- **FileNumber [4 bytes]**: File Number of the file containing the block to prefetch
- **BlockNumber [4 bytes]**: Block Number of the file block to prefetch

b. **Common Block Subsection:**

- **NumCommonBlocks [4 bytes]:** Number of critical blocks.

**Block Structure: (variable number of entries)**

- **FileNumber [4 bytes]**: File Number of the file containing the block to prefetch
- **BlockNumber [4 bytes]**: Block Number of the file block to prefetch

## 5. Profile Section:

The profile section consists of a reference sequence of file blocks accessed by the application at runtime. Conceptually, the profile data is a two dimensional matrix. Each entry [*row, column*] of the matrix is the frequency a block *row* is followed by a block *column*. In any realistic applications of fair size, this matrix is very large and sparse. Proper data structure must be selected to store this sparse matrix efficiently in required storage space and minimize the overhead in accessing this data structure access.

The section is constructed from two basic structures: row and column structures. Each row structure is followed by N column structures specified in the NumberColumns field. Note that this is an optional section. But with appropriate profile data, the SAS client prefetcher performance can be increased.

**Row Structure: (variable number of entries)**

- **FileNumber [4 bytes]**: File Number of the row block
- **BlockNumber [4 bytes]**: Block Number of the row block
- **NumberColumns [4 bytes]**: number of blocks that follows this block. This field determines the number of column structures following this field.

**Column Structure: (variable number of entries)**

- **FileNumber [4 bytes]**: File Number of the column block
- **BlockNumber [4 bytes]**: Block Number of the column block
- **Frequency [4 bytes]**: frequency the row block is followed by column block

## 6. Comment Section:

The comment section is used by the Builder to describe this AppInstallBlock in more detail.

- **CommentLengthIsAnsi [1 byte]**: 1 if string is ansi, 0 if Unicode format.
- **CommentLength [4 bytes]**: Byte size of the comment string
- **Comment [X bytes]**: Null terminating comment string

## 7. Code Section:

The code section consists of the application-specific initialization code needed to run on the SAS client to setup the client machine for this particular application. This section may be empty if the default initialization procedure in the SAS client is able to setup the client machine without requiring any application-specific instructions. On the Windows system, the code is a DLL file containing two exported function calls: *Install*(), *Uninstall*(). The SAS client loads the DLL and invokes the appropriate function calls.

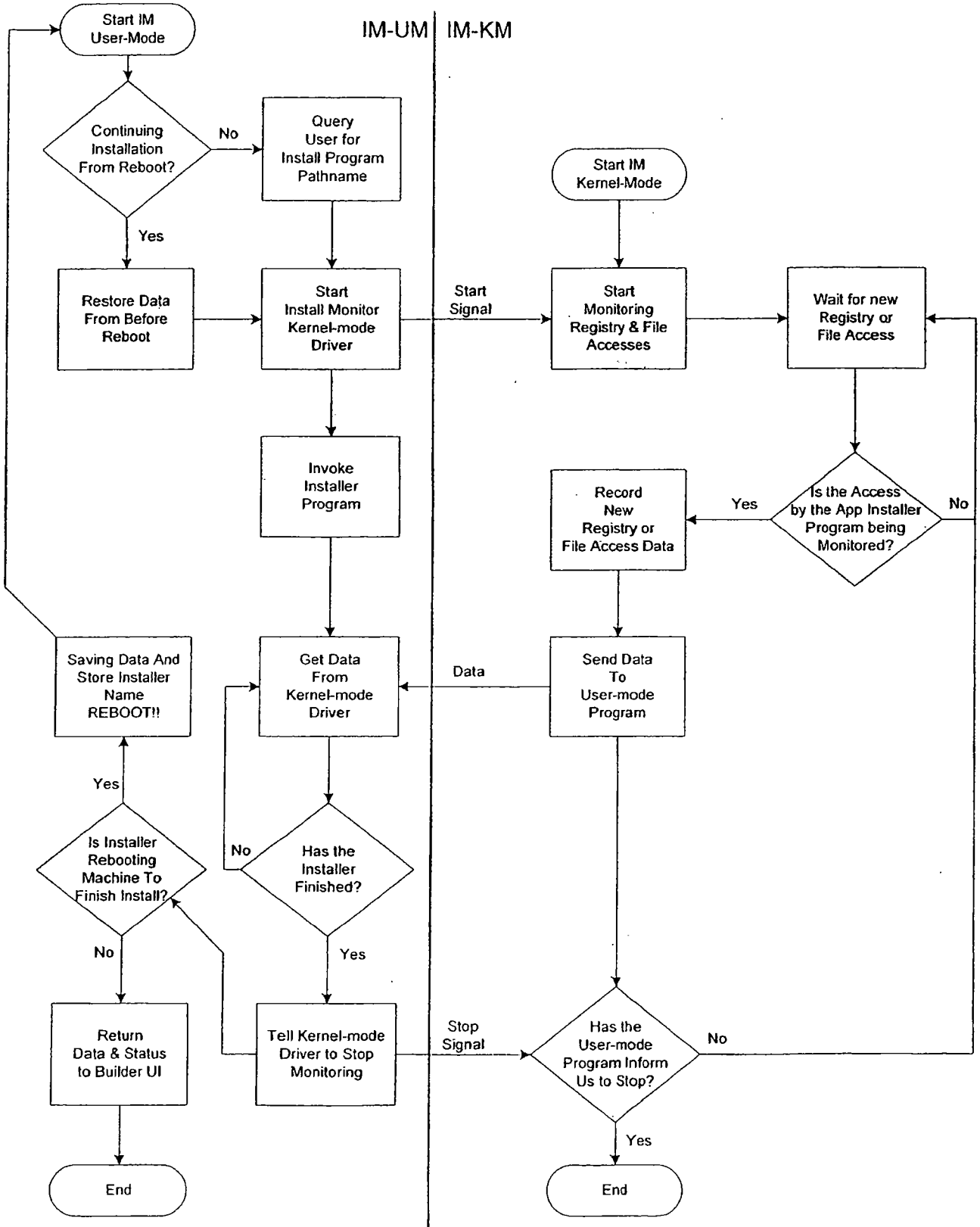
- **CodeLength [4 bytes]**: Byte size of the code
- **Code [X bytes]**: Binary file containing the application-specific initialization code. On Windows, this is just a DLL file.

## 8. LicenseAgreement Section:

The Builder creates the license agreement section. The SAS client displays the license agreement text to the end-user before the application is started for the first time. The end-user must agree to all licensing agreement set by the software vendor in order to use the application.

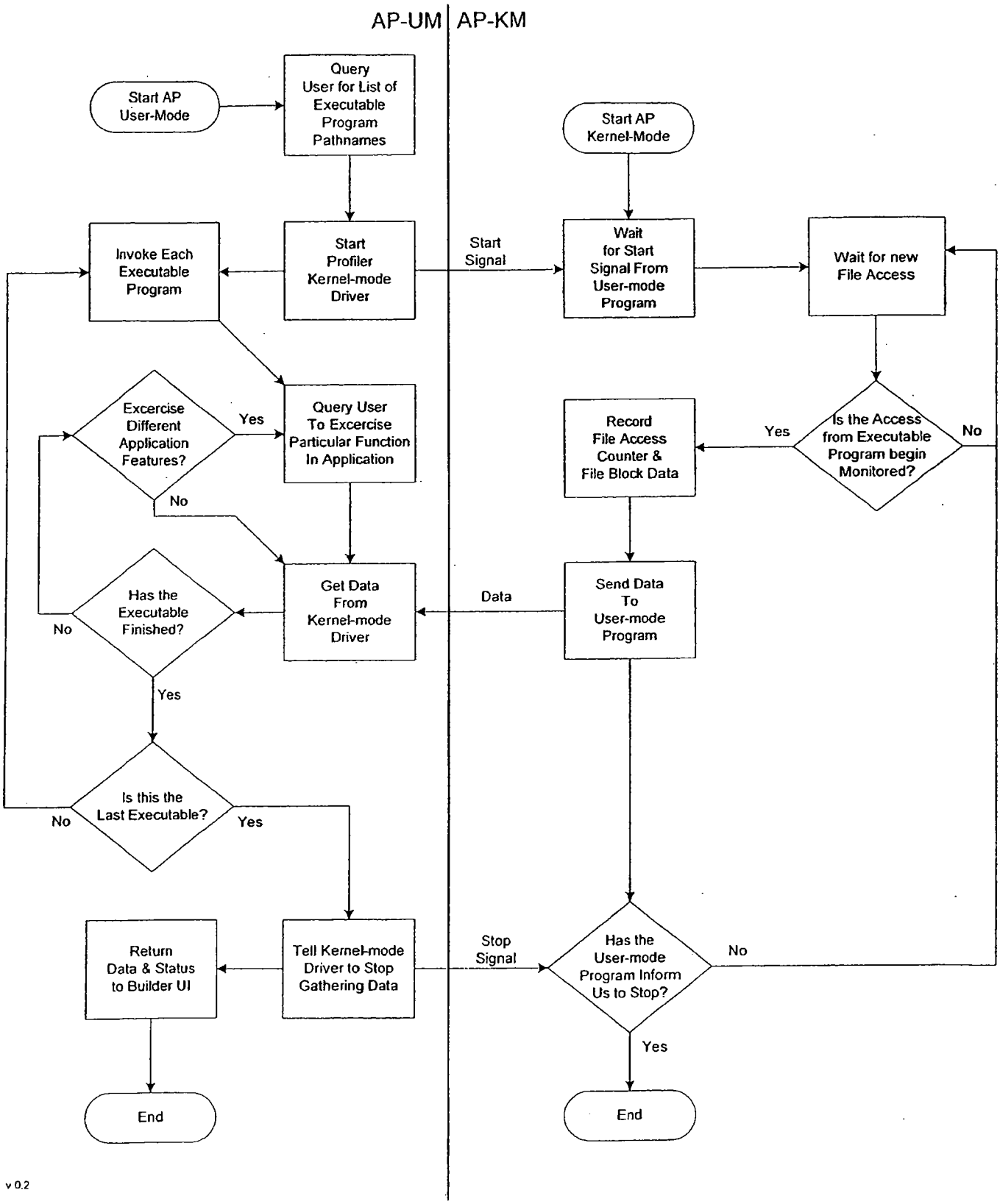
- **LicenseTextIsAnsi [1 byte]**: 1 if ansi, 0 if Unicode format.
- **LicenseTextLength [4 bytes]**: Byte size of the license text
- **LicenseAgreement [X bytes]**: Null terminating license agreement string

## Figure 1: Builder Install Monitor (IM) Control Flow Diagram



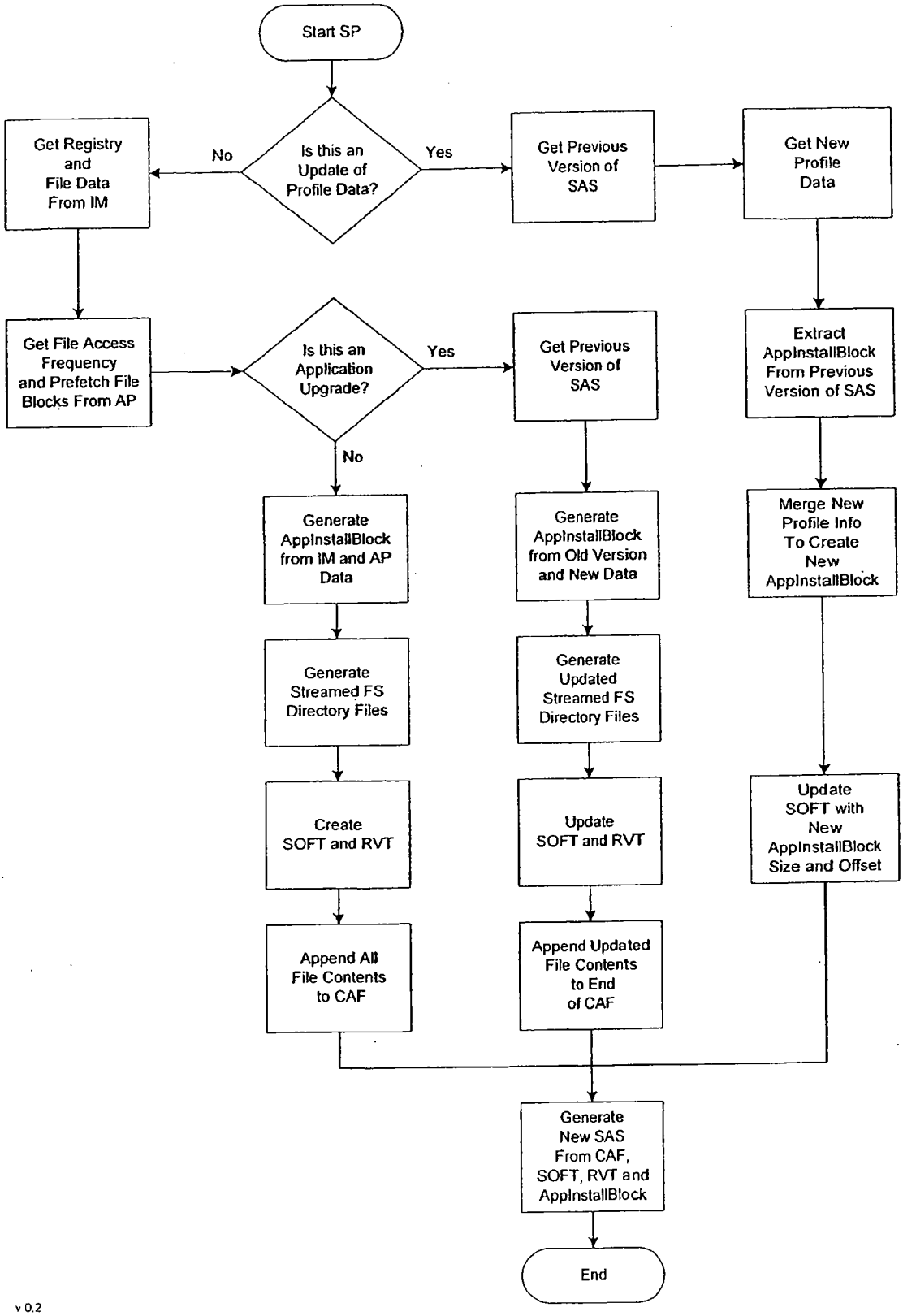

v 0.2

## Figure 2: Builder Application Profiler (AP) Control Flow Diagram



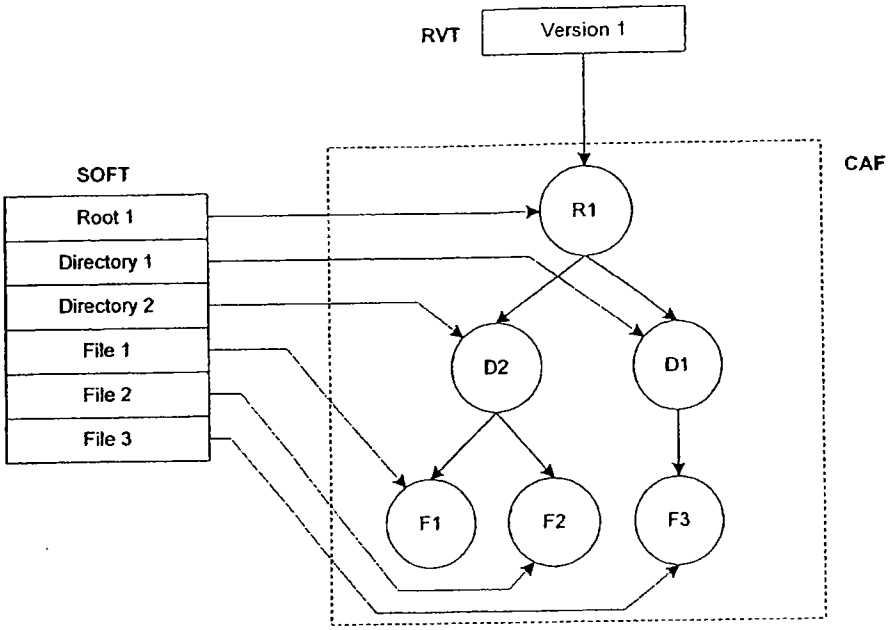

v 0.2

## Figure 3: Builder SAS Packager (SP) Control Flow Diagram

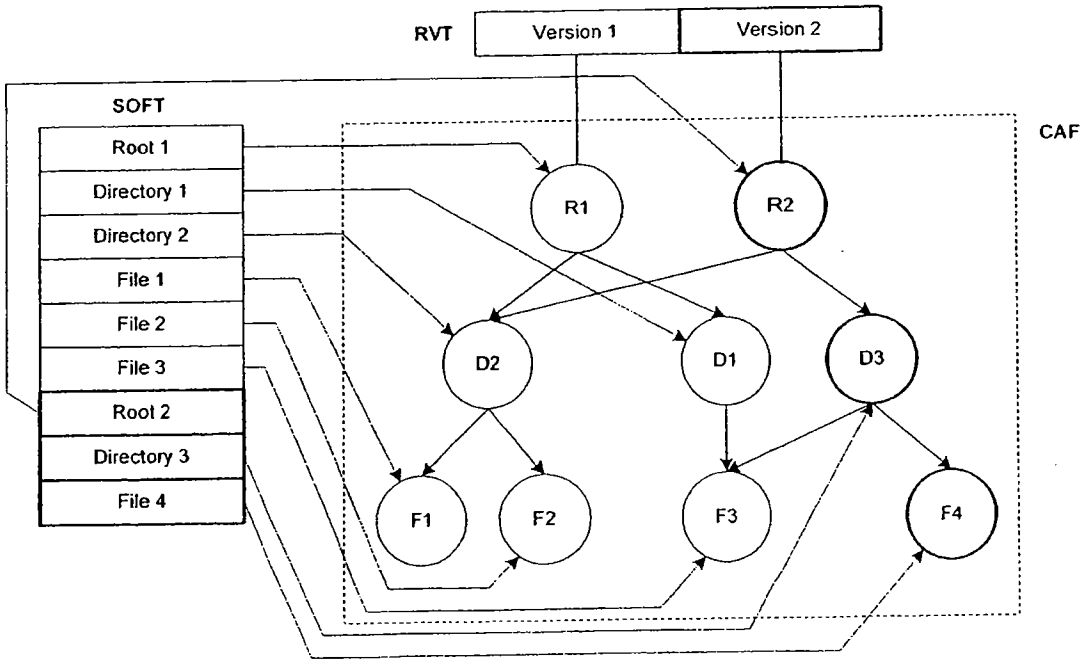



v 0.2

## Figure 4: Internal Representation of SAS with Support for Versioning

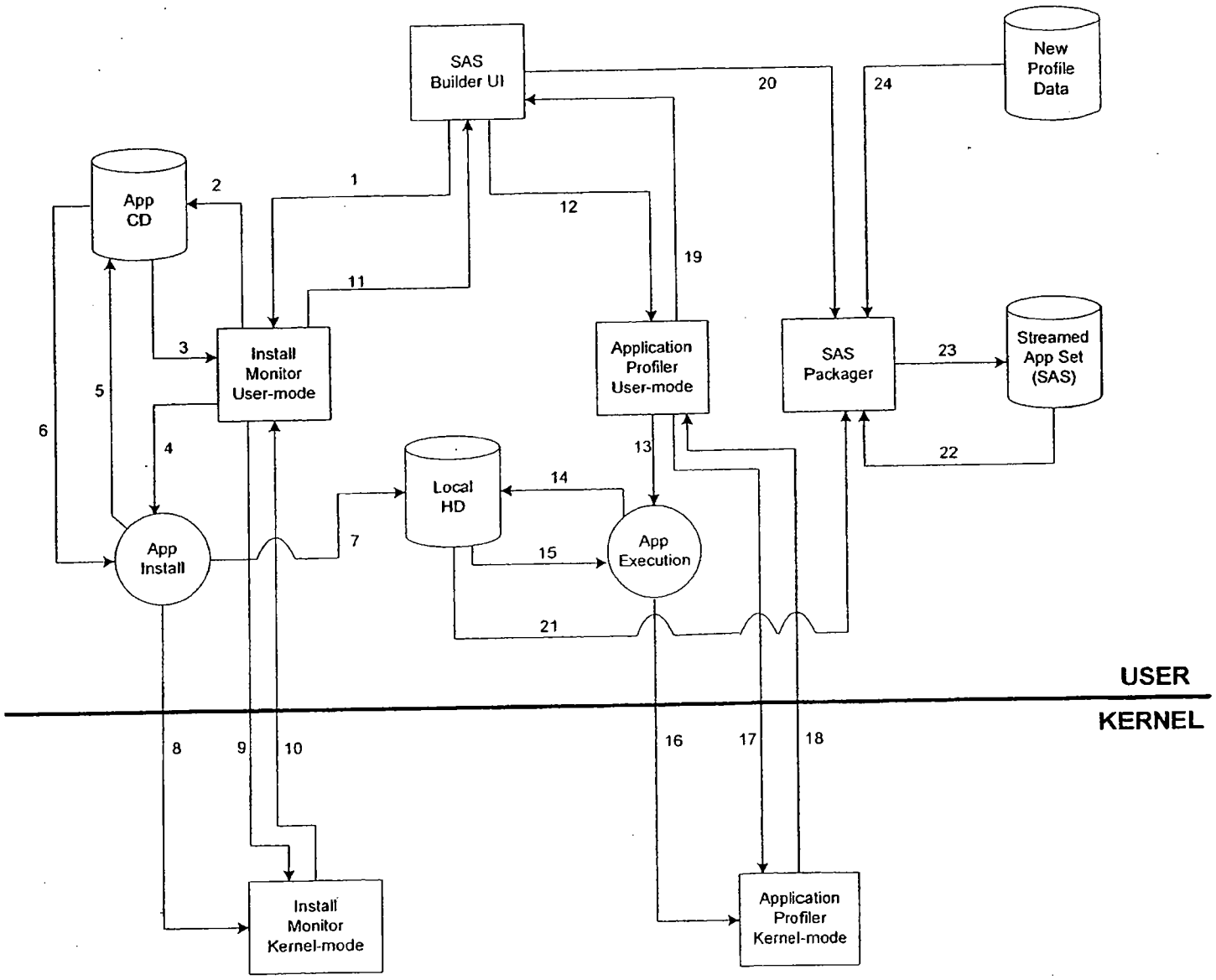



Version 1 of Application in SAS



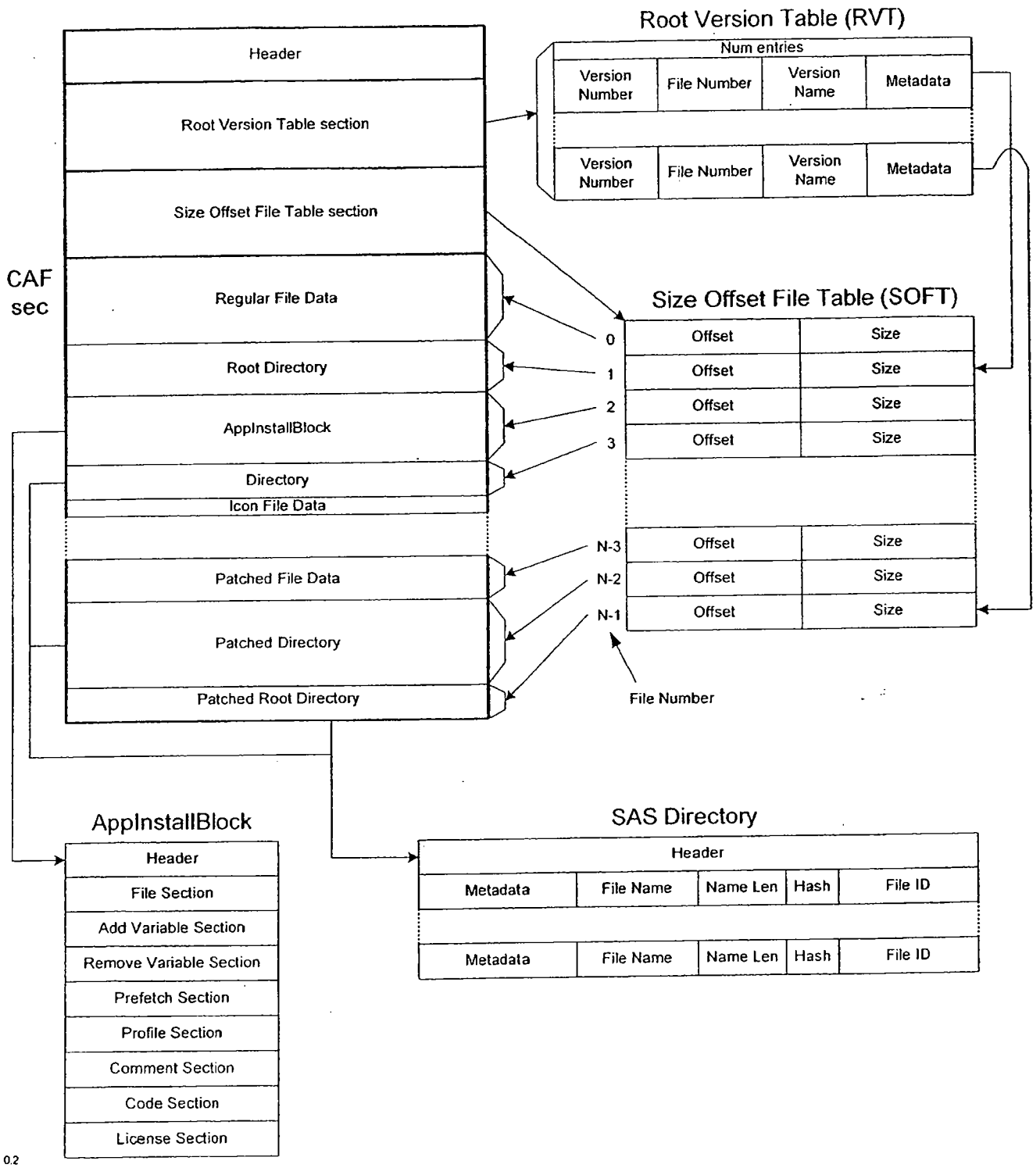

Version 2 of Application in SAS with a New File

## Figure 5: Streamed Application Set (SAS) Builder Data Flow Diagram




v 0.2

## Figure 6: Content of a Streamed Application Set




v 0.2

Figure 7: Device Driver Paradigm

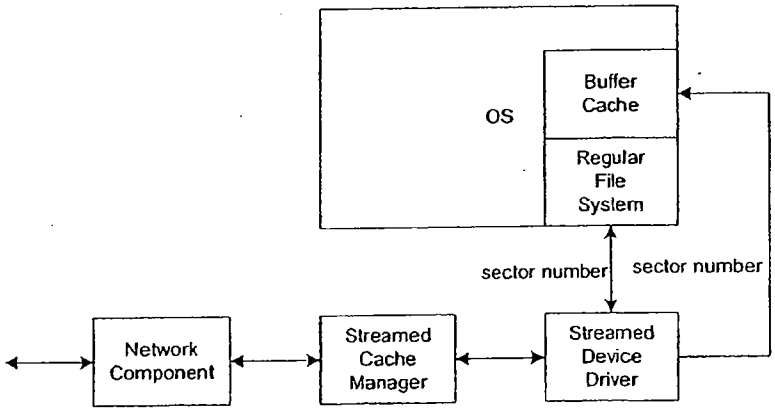
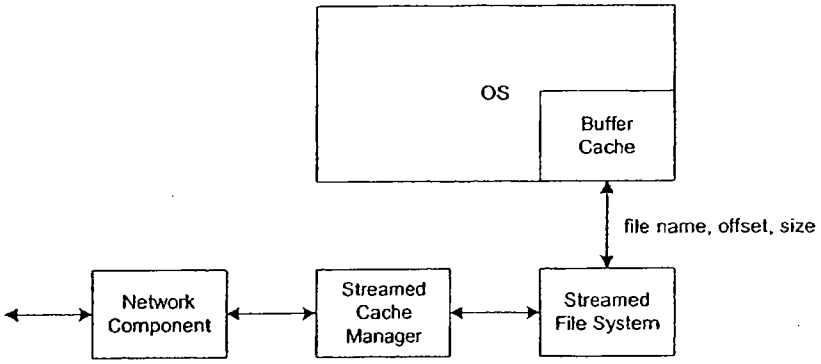
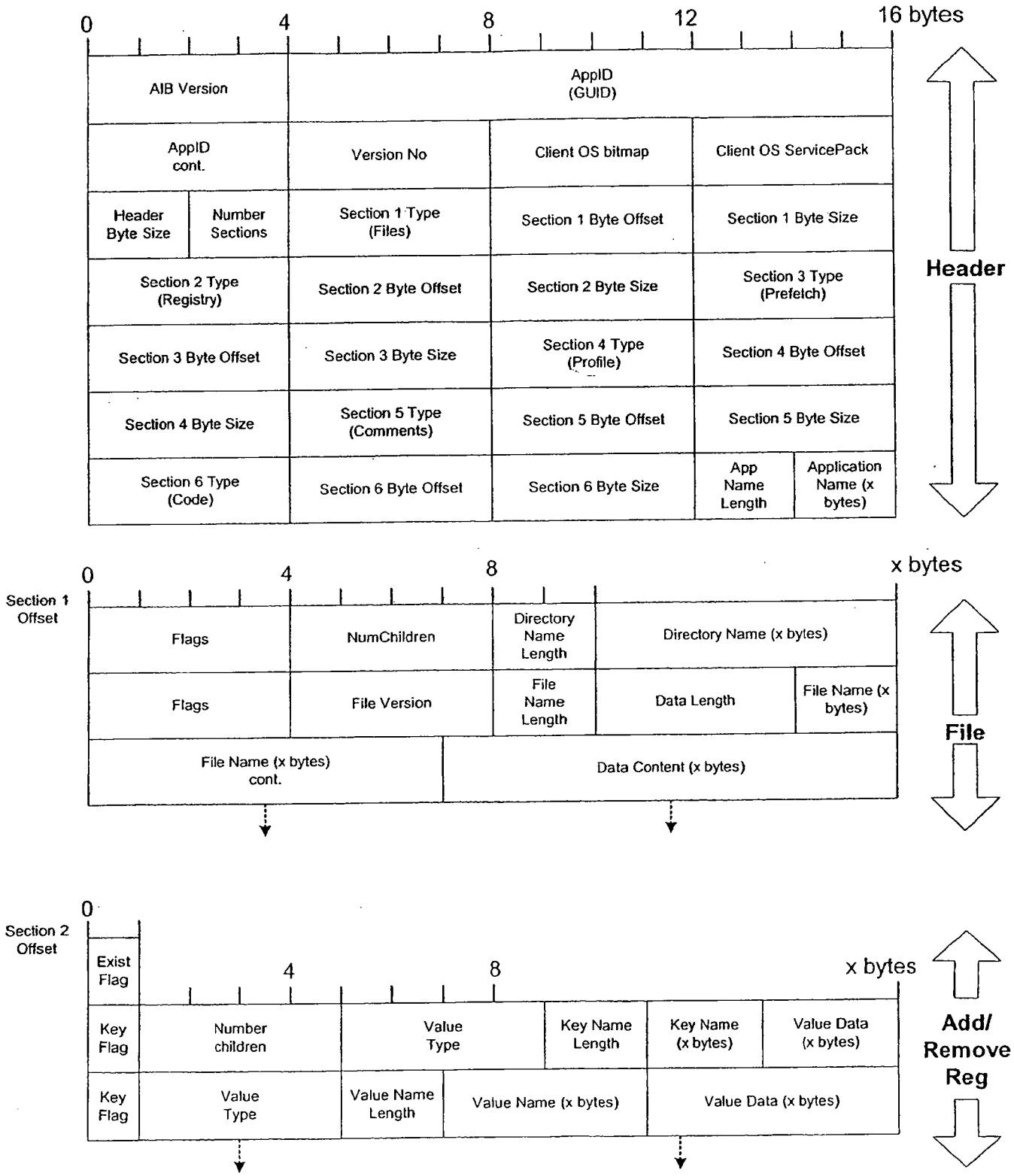


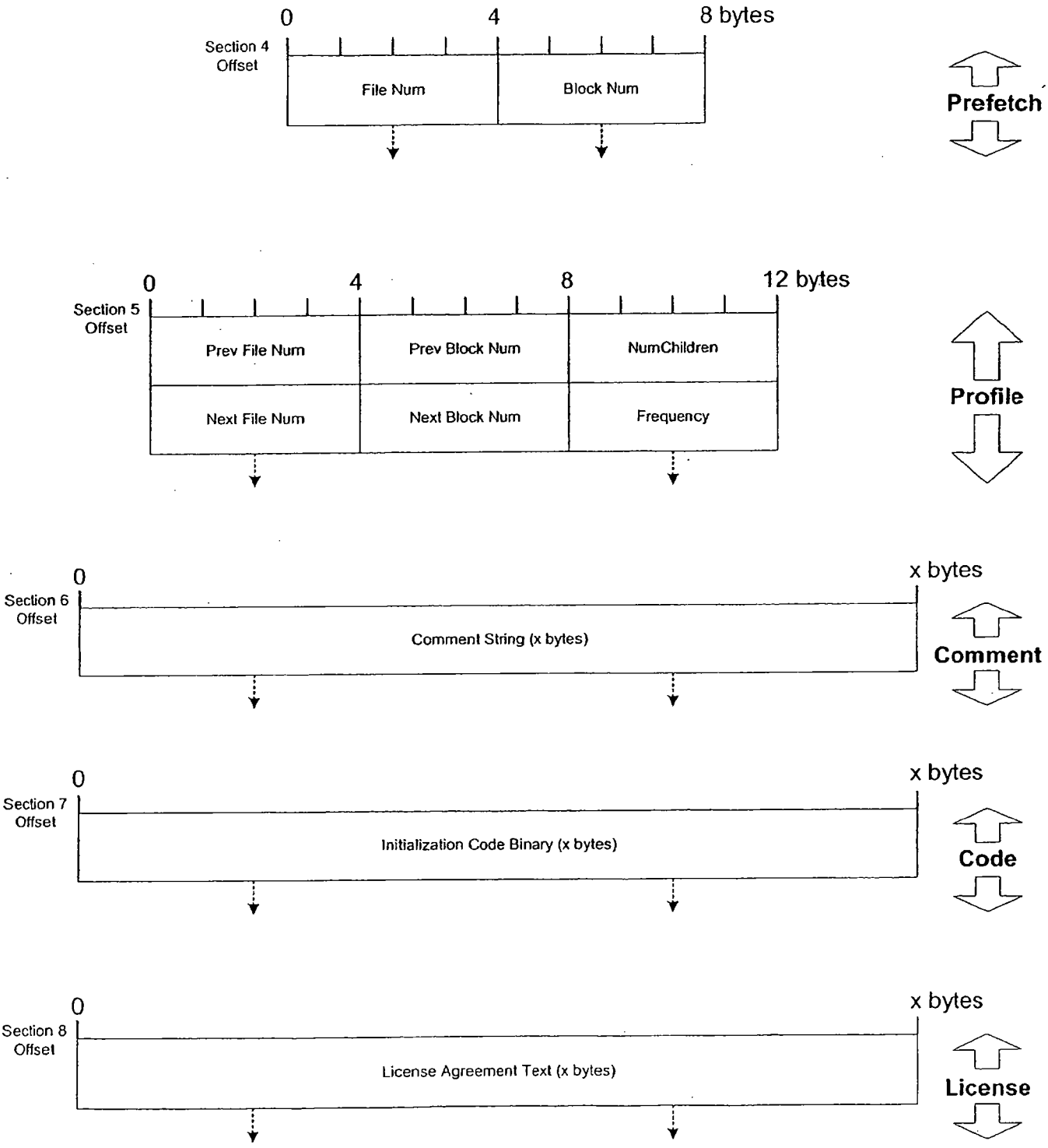
Figure 8: File System Paradigm



## Figure 10: Format of AppInstallBlock (part 1 of 2)



# Format of AppInstallBlock (part 2 of 2)



# eStream Application Builder High-Level Design

Authors: Sanjay Pujare and David Lin

Version 0.1

[redacted]

This document contains the high level design of the eStream Application Builder. The Builder is used to "prepare" an application before it can be eStreamed. This document describes the high level design of the application installation monitoring, file relocation and mapping, gathering of the initial profiling information of an application, the packaging of the eStream Set, and the merging of the newly uploaded user profile data.

Note: all references to "user" should be understood to mean the user of the Builder (i.e. the person who is responsible for creating eStream sets) and not the end-user of eStream technology.

This document described these steps involved in the preparation of the application: Installation Monitoring, Application Profiling, and eStream Packaging.

## *Modules*

**Installation Monitor:**

- When the application is installed, we need to monitor the installation to see various "things" taking place on the computer. These could be:
  - Various updates to the System Registry
  - Files added to the Install directories (i.e. directories where application bits are copied as specified by the installing user). Lets call this group $F_I$.
  - Files added/updated to the Shared directories (e.g. "Program Files\Common Files"). Lets call this group $F_C$.
  - Files added/updated to the System directories (e.g. "WinNT\System32"). Lets call this group $F_S$.
  - Files added/updated to the User specific directories (e.g. "Documents and Settings\spujare\Application Data"). Lets call this group $F_U$.

  Note that once this information is gathered by the "Installation Monitor", a single "Installation Set" is prepared where all the files are stored in a single directory hierarchy. Note that files in the $F_C$, $F_S$ and $F_U$ groups (i.e. $F_{CSU}$ group) are also stored here. For these files a "mapped location" is created under the single directory hierarchy. The Installation Set typically creates a map of all files (called ISM for Installation Set Map) described above with each entry containing the following info:
  1. fileId for the file
  2. location and name of the file. Note that the location will be the actual location for $F_I$ files, but mapped location for the $F_{CSU}$ files.

- After we gather the above information, we need to prepare a "File Relocation Map" (FRM) that is used by the client file spoofer to spoof references to any file in the common file group (i.e. $F_{CSU}$). For example: when the eStreamed app makes a reference to a file `C:\Program Files\Word\Foobar`, the file spoofer actually redirects that reference to `Z:\Program Files\Word\Foobar`. It does that because of the File Relocation Map. Each entry in the FRM typically has the following info:
  1. fileId (which references an entry in the ISM).
  2. Actual location where the application expects it (i.e. `C:\Program Files\Word\Foobar`).

**Profile Module:**
During the application building process, the Builder program queries the user for the name of the application executable. Then Builder program starts and terminates the application executable immediately to gather initial sequence of the application page access pattern. After the initial seed of profile data is acquired, the Profile Sequence Matrix is combined with other appInstallBlock data gathered from the Install Monitor.

Profile Sequence Matrix is a 2D matrix of a profile data. Each entry of the matrix [column C, row R] is an integer value indicating the number of times a page R is requested following the request of page C. This successor request pattern is the page requests missed in the eStream cache manager.

**Package Module:**
In the final phase of the Builder program, the appInstallBlock is encapsulated into a special installation executable and the application files is archived into a single compressed package. The install executable containing the appInstallBlock and the archive of application files can then be placed in a suitable eStream Set server for ASP to download to their machines.

**Merge Module: (not supported in version 1.0)**
During normal eStream application usage, the eStream client gathers profile information for that particular run of the application. Then at the termination of an eStream application, it uploads the new Profile Sequence Matrix to the Profile Server. The clients should not upload the Profile Sequence Matrix from previous runs because the Profile Server has no mechanism for distinguishing between previously uploaded data and the newly acquired data.

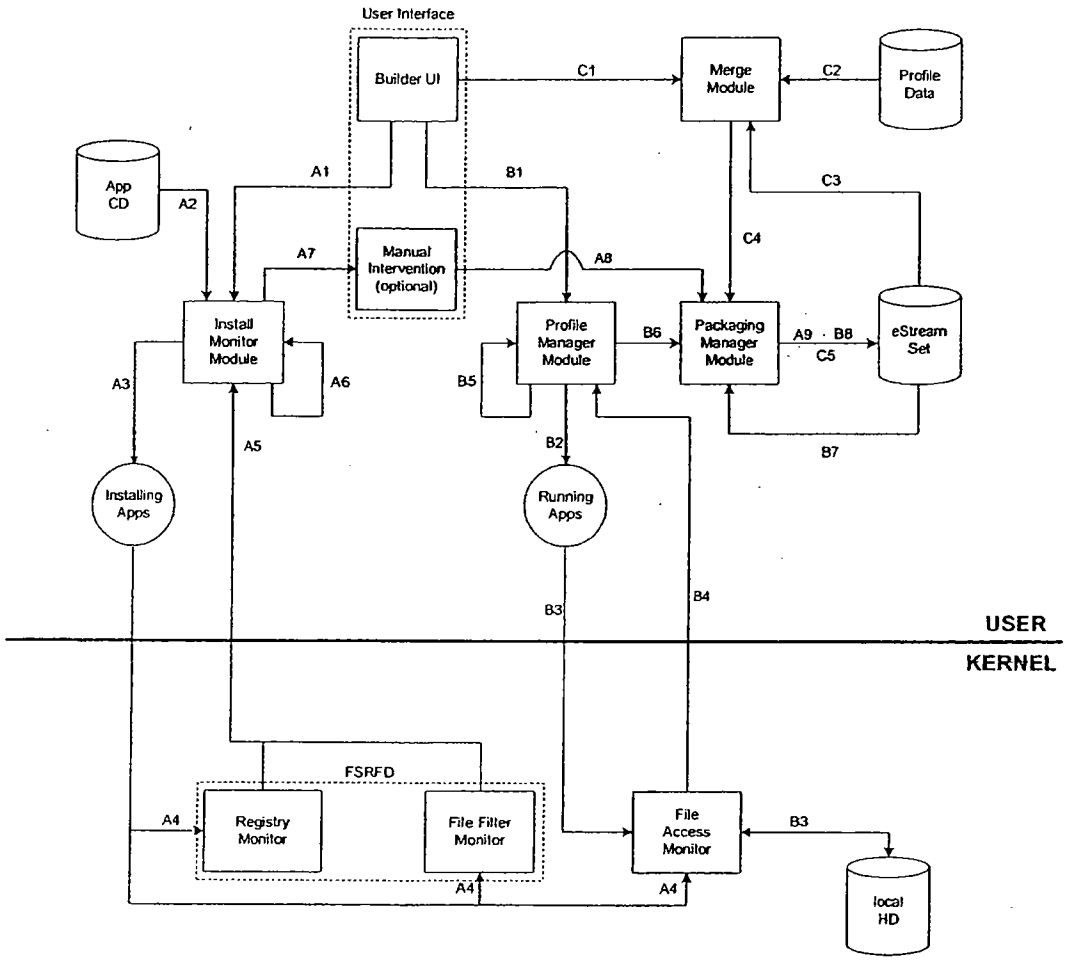
At appropriate time, the Builder is invoked to merge the newly uploaded per-user Profile Sequence Matrix into a collective Matrix. The merging algorithm may be designed with some heuristics to prevent the data biasing toward power users. This collective Matrix can be reinserted into the appropriate appInstallBlock then downloaded by any requesting eStream clients.

**Kernel Device Drivers:**

In addition, kernel device drivers are used to actually hook into the operating system to monitor the registry and file changes during installation of the application. This is accomplished by the FSRFD module.

The kernel device driver is also used for gathering monitoring file block references from the operating system to the file system. This is accomplished by the File Access Monitor.

# eStream Application Builder High-Level Design Diagram




v 0.1

## Interfaces

The interfaces are divided into three use cases: application installation monitoring, application profiling, and merging of the uploaded profile data.

**Use Case #1: Install Monitor**

A1. Builder UI to Install Monitor – send the name of the application installation executable
A2. App CD to Install Monitor – the CD containing the application is fed into the installation monitor module
A3. Install Monitor to Installation App – invoke the installation program
A4. Installation App to FSRFD – monitor all changes to the registry and files when installation program write to local file system
A5. FSRFD to Install Monitor – send all registry and file changes
A6. Install Monitor to itself – repeat all applications in the suite and merge all data
A7. Install Monitor to Manual Intervention – send a list of registry and file captured by the install monitor to UI and allow user to add or delete any entries
A8. Manual Intervention to Package Manager – send the final registry and file relocation data to the packager
A9. Package Manager to database – data set is packaged into appInstallBlock and the rest of the application files suitable for eStreaming

**Use Case #2: Profiling**

B1. Builder UI to Profile Manager – send the name of the application executable
B2. Profile Manager to Run App – invoke the application
B3. Run App to eStream File Access Monitor – record sequences of page requests
B4. File Access Monitor to Profile Manager – save the profile information
B5. Profile Manager to Profile Manager – repeat for each application in the suite
B6. Profile Manager to Package Manager – send all profile data for merging into a single data
B7. database to Package Manager – get eStream Set from the database
B8. Package Manager to database – save the updated eStream Set

**Use Case #3: Merging Profile data (not supported in version 1.0)**

C1. Builder UI to Merger – send the application name with profile data to merge
C2. database to Merger – get uploaded Profile Sequence Matrix from the Profile Server
C3. database to Merger – get the old appInstallBlock from database
C4. Merger to Package Manager – reinsert the Profile data into appInstallBlock
C5. Package Manager to database – save the updated appInstallBlock

## Requirements

Please see eStream1.0-REQ.doc for the most up-to-date list of the Builder requirements. This requirement list may not contain the most recent changes. Each requirement is identified by a tag such as R-XXXX for easy references elsewhere in the document.

- **R-Background**: The installation monitor runs in the background, when an eStream application is installed as part of its preparation or building.
- **R-RegistryCapture**: The installation monitor captures all the updates to the System Registry that take place during the install. These updates are captured as a .REG file. Note that registry key deletions are also captured and stored in the .REG file. Please see the LLD doc by Charles Booher about the Registry spoofing database.
- **R-FileCapture:** The installation monitor records all the files created in the two kinds of directories: the install directory (the $F_I$ group described above) and the common directories (the $F_{CSU}$ group). All the files created are copied to the Installation Set and the File Relocation Map (FRM) created for the $F_{CSU}$ group files. <Note: as far as a system common DLL is concerned, the eStream client should (a) overwrite the existing DLL if it exists (b) spoof it if doesn't exist. This is necessary because some installations may depend on newer versions of, say MSVCRT.DLL and in Windows there is no way to maintain different versions of the same DLL>.
- **R-InitialProfiling:** The Builder must be able to gather initial set of application profile data. This data consists of the page access pattern for starting and immediately shutting down an application.
- **R-Packaging:** The Builder must package the eStream Set into a easily manageable packages suitable for ASP administrators to download to their servers. The package can be divided into two sets:

    1. Installation Set - an appInstallBlock which is a set of data needed to setup the client machine for running a particular eStream application. The appInstallBlock is converted into an installation executable for simplifying the initial application set-up on the client machine.
    2. Run-time Set - a set of files associated with a particular application. At run-time, appropriate pages from this set of files is streamed to the client.

- **R-Merging (not supported in version 1.0):** The Builder must be able to collect per-user profile data from the Profile Server and merge the profile data into a combined data usable for updating the profile data in the appInstallBlock. This profile data can also be collected for use by the ASP or application developers.
- **R-NoQuietOperation:** The Builder is not required to be run in an environment where no other applications are running. But, since the Builder operates by invoking application installation program, it inherits any restrictive "Quiet-Operation" requirement from the installation program. Thus, if the installation program of an application has a "Quiet-Operation" requirement, then the "Quiet-Operation" must be enforced by the user when running the Builder.
- **R-AllClient**: The Builder should provide functionality to create installation set(s) for each of the clients eStream 1.0 is going to support. <Preferably there should be only one builder program that should recognize the OS it is running on and should create the appropriate installation set. Also if possible, we should be able to "diff" installation sets for different OSs and if they are same, we should be able to create

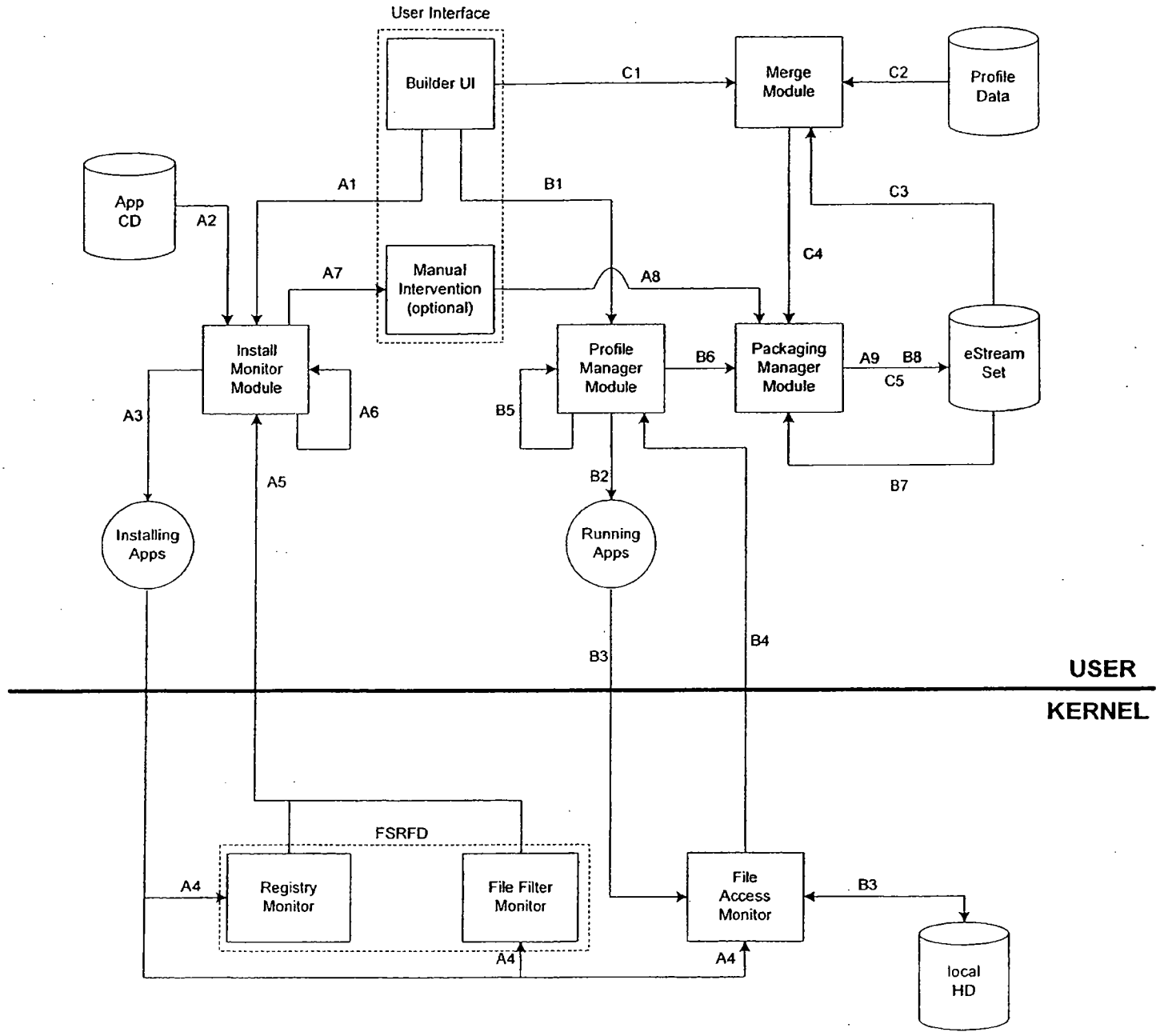
a single installation set for those OSs. The clients to be supported are W2K, WinNT4.0 and Win98>.

- **R-AppIdGeneration**: It should be possible to change the appId of the eStream set when an ASP wants to "install" the eStream set in order to host it. Typically the builder will generate a default appId number for a new application which can be overridden by the ASP installer by using a Builder tool.
- **R-SuiteSupport**: It should be possible to create a merged eStream set for a suite of applications. E.g. Office consisting of Word, Excel and Powerpoint. This could be done either by providing a tool for merging multiple eStream sets or by allowing the builder to serially monitor multiple installations in a session and then allowing the user to create a single package at the end of the session.
- **R-Testing**: It should be possible to test the Builder using a stand-alone tester and not require the eStream client+server programs.
- **R-UpgradeSupport**: The appInstallBlock should have support for indicating upgrades at the support site. E.g. When an eStream application is upgraded at the server (not as a separate app), the client will no longer be able to access/use it. We should provide some version of the appInstallBlock itself so that clients should detect that they will need to download the appInstallBlock again.
- **R-ManualIntervention**: In the process of creating an eStream set it should be possible for the user to delete file entries and registry entries manually to "trim" the eStream set if she so desires assuming the user knows what she is doing.

## *Issues*

- Profile Sequence Matrix is different for different machine configuration even if the user's usage pattern is the same.
- Profile Sequence Matrix doesn't contain the right successor profile information as eStream cache is warmed up and pages from the cache is replaced.
- Merging Module must take different machine configuration into account. Should this information be uploaded by the client as the same time it uploads the Profile Sequence Matrix to the Profile Server?
- What is the difference between profiling based on the page sequencing seen by the eStream Cache Manager versus the page sequencing missed by the eStream Cache Manager?

# eStream Application Builder High-Level Design Diagram



# Tricky Builder Issues

## Author: Sanjay Pujare

This document enumerates all those tricky issues that may make the Builder's job difficult. Even though some solution may be proposed for some issues, not every issue would have a solution described in this document. The purpose of this document is mainly to keep track of Builder issues that may impose some limitations on the eStream technology. This way Omnishift marketing and deployment are aware of these limitations.

1) The Builder cannot capture updates to existing files in an intelligent fashion (i.e. if the updates are based on a context or existing contents, it is very difficult to capture that). So the current Builder will just flag an error, if such an update occurs.

   **Solution**
   - These updates are probably very rare, so we can defer it to the next release.
   - For this release, we can try to solve this on a case by case basis e.g. we will try to solve this issue for INI files.
   - Based on our understanding of general app installations, we might be able to make some generalizations that we can use in eStream e.g. Only certain files get updated; there is a definite pattern of updates.

2) The current Builder drivers are based on the NT driver model, and hopefully we can implement the same functionality in the Win98 drivers, but this needs to be ensured. (This shouldn't be an issue, but...)
3) We need to think some more about those cases when device drivers are installed by apps. Issues that can arise:
   - This may not work correctly on the eStream client, just because the driver installation didn't take place properly.
   - The Builder would need to be able to figure out in an automated way if a client reboot is required or not.
   - If the driver installation is h/w or s/w specific that can be difficult to tackle.

   **Solution**
   - As we eStream more apps and gain more experience, we should be able to figure out solutions.
4) There could be an ambiguity when the Installmon is trying to change absolute paths (or absolute values in general) to relative paths. e.g. A path like C:\WINNT can be changed to %SystemRoot% or %windir% since both of those environment variables are set to "C:\WINNT" on my system.

   **Solution**
   - We can prioritize env vars and registry keys as described in the BuilderUI-LLD design document.

- We should encourage Builder operators to use as distinct values as possible for env variables and registry keys for the Builder machine.

# eStream Set Format Low Level Design

*Sanjay Pujare and David Lin*
*Version 0.7*

## Functionality

The eStream Set is a data set associated with an application suitable for streaming over the network. The eStream Set is generated by the eStream Builder program. This program converts locally installable applications into the eStream Set. This document describes the format of the eStream Set.

**Note:** Fields greater than a single byte is stored in little-endian format. The eStream Set file size is limited to 2^64 bytes. The files in the CAF section are laid out in the same order as its corresponding entries in the SOFT table.

## Data type definitions

The format of the eStream Set consists of 4 sections: header, Root Version Table (RVT), Size Offset File Table (SOFT), and Concatenation Application File (CAF) sections.

### 1. Header section

- **MagicNumber [4 bytes]:** Magic number identifying the file content with the eStream Set
- **ESSVersion [4 bytes]:** Version number of the eStream Set format.
- **AppID [16 bytes]:** A unique application ID for this application. This field must match the AppID located in the AppInstallBlock. Guidgen is used to create this identifier.
- **Flags [4 bytes]:** Flags pertaining to EStreamSet
- **Reserved [32 bytes]:** Reserved spaces for future.

- **RVToffset [8 bytes]:** Byte offset into the start of the RVT section.
- **RVTsize [8 bytes]:** Byte size of the RVT section.
- **SOFToffset [8 bytes]:** Byte offset into the start of the SOFT section.
- **SOFTsize [8 bytes]:** Byte size of the SOFT section.
- **CAFoffset [8 bytes]:** Byte offset into the start of the CAF section.
- **CAFsize [8 bytes]:** Byte size of the CAF section.

- **VendorNameIsAnsi [1 byte]:** 0 if the vendor name is in Unicode format. 1 if the vendor name is in ANSI format.
- **VendorNameLength [4 bytes]:** Byte length of the vendor name.
- **VendorName [X bytes]:** Name of the software vendor who created this application. I.e. "Microsoft". Null-terminated.

- **AppBaseNameIsAnsi [1 byte]:** 0 if the vendor name is in Unicode format. 1 if the vendor name is in ANSI format.
- **AppBaseNameLength [4 bytes]:** Byte length of the application base name.
- **AppBaseName [X bytes]:** Base name of the application. I.e. "Word 2000". Null-terminated.
- **MessageIsAnsi [1 byte]:** 0 if the vendor name is in Unicode format. 1 if the vendor name is in ANSI format.
- **MessageLength [4 bytes]:** Byte length of the message text.
- **Message [X bytes]:** Message text. Null-terminated.

## 2. Root Version Table (RVT) section

The Root version entries are ordered in a decreasing value according to their file numbers. The Builder generates unique file numbers within each eStream Set in a monotonically increasing value. So larger root file number implies later versions of the same application. The latest root version is located at the top of the section to allow the eStream Server easy access to the data associated with the latest root version.

- **NumberEntries [4 bytes]:** Number of patch versions contained in this eStream Set. The number indicates the number of entries in the Root Version Table (RVT).

**Root Version structure: (variable number of entries)**

- **VersionNumber [4 bytes]:** Version number of the root directory.
- **FileNumber [4 bytes]:** File number of the root directory.
- **VersionNameIsAnsi [1 byte]:** 0 if the vendor name is in Unicode format. 1 if the vendor name is in ANSI format.
- **VersionNameLength [4 bytes]:** Byte length of the version name
- **VersionName [X bytes]:** Application version name. I.e. "SP 1".
- **Metadata [32 bytes]:** See eStream FS Directory for format of the metadata.

## 3. Size Offset File Table (SOFT) section

The SOFT table contains information to locate specific files in the CAF section. The entries are ordered according to the file number starting from 0 to NumberFiles-1. The start of the SOFT table is aligned to 8 bytes boundary for faster access.

**SOFT entry structure: (variable number of entries)**

- **Offset [8 bytes]:** Byte offset into CAF of the start of this file.

- **Size [8 bytes]:** Byte size of this file. The file is located from address Offset to Offset+Size.

# 4. Concatenation Application File (CAF) section

CAF is a concatenation of all file or directory data into a single data structure. Each piece of data can be a regular file, an AppInstallBlock, an eStream FS directory file, or an icon file.

## *a. Regular Files*

- **FileData [X bytes]:** Content of a regular file

## *b. AppInstallBlock (See AppInstallBlock document for detail format)*

A simplified description of the AppInstallBlock is listed here. For exact detail of the individual fields in the AppInstallBlock, please see AppInstallBlock Low-Level Design document.

- **Header section [X bytes]:** Header for AppInstallBlock containing information to identify this AppInstallBlock.
- **Files section [X bytes]:** Section containing file to be copied or spoofed.
- **AddVariable section [X bytes]:** Section containing system variables to be added.
- **RemoveVariable section [X bytes]:** Section containing system variables to be removed.
- **Prefetch section [X bytes]:** Section containing pointers to files to be prefetched to the client.
- **Profile section [X bytes]:** Section containing profile data. (not used in eStream 1.0)
- **Comment section [X bytes]:** Section containing comments about AppInstallBlock.
- **Code section [X bytes]:** Section containing application-specific code needed to prepare local machine for streaming this application
- **LicenseAgreement section [X bytes]:** Section containing licensing agreement message.

## *c. EStream Directory*

An eStream Directory contains information about the subdirectories and files located within this directory. The information includes file number, names, and metadata associated with the files.

- **MagicNumber [4 bytes]:** Magic number for eStream directory file.
- **ParentFileID [16+4 bytes]:** AppID+FileNumber of the parent directory. AppID is set to 0 if the directory is the root.

- **SelfFileID [16+4 bytes]:** AppID+FileNumber of this directory.
- **NumFiles [4 bytes]:** Number of files in the directory.
- **NumEntries [4 bytes]:** Number of entries in the directory. Some entries are used for storing long file names and some are unused due to deleted files. So the NumEntries must be equal or less than NumFiles.

Fixed length entry for each file in the directory consists of 2 formats (short format for storing files with name that fit the 8.3 convention; and long format for storing long file names). Each entry is 84 bytes and the entry are aligned on every 4K page boundry. Thus, in the first 4K page of the directory, the padding consists of 12 unused bytes (52 bytes for header + 48 entries * 84 bytes per entry + 12 unused bytes = 4096 bytes). In all subsequent pages, the padding is 64 bytes (48 entries * 84 bytes per entry + 64 unused bytes = 4096 bytes):

**Short Filename entry:**

- **Format [1 byte]:** Format of this entry, should be 's' for short format, 'l' for long filename format, or possibly 'u', for unused.
- **ShortLen [1 byte]:** Length of short file name.
- **LongLen [1 byte]:** Length of long file name.
- **UNUSED [1 byte]:** Padding
- **NameHash [4 bytes]:** Hash value of the short file name. Algorithm TBD.
- **ShortName [24 bytes]:** 8.3 short file name in unicode
- **FileID [16+4 bytes]:** AppID+FileNumber of each file in this directory.
- **Metadata [32 bytes]:** The metadata consists of file **byte size** (8 bytes), file **creation time** (8 bytes), file **modified time** (8 bytes), **attribute flags** (4 bytes), **eStream flags** (4 bytes). The bits of the **attribute flags** have the following meaning:
  - **Bit 0:** Read-only – Set if file is read-only
  - **Bit 1:** Hidden – Set if file is hidden from user
  - **Bit 2:** Directory – Set if the file is an eStream Directory
  - **Bit 3:** Archive – Set if the file is an archive
  - **Bit 4:** Normal – Set if the file is normal
  - **Bit 5:** System – Set if the file is a system file
  - **Bit 6:** Temporary – Set if the file is temporary

  The bits of the **eStream flags** have the following meaning:
  - **Bit 0:** ForceUpgrade – Used only on root file. Set if client is forced to upgrade to this particular version if the current root version on the client is older.
  - **Bit 1:** RequireAccessToken – Set if file require access token before client can read it.
  - **Bit 2:** Read-only – Set if the file is read-only

**Long Filename entry:**

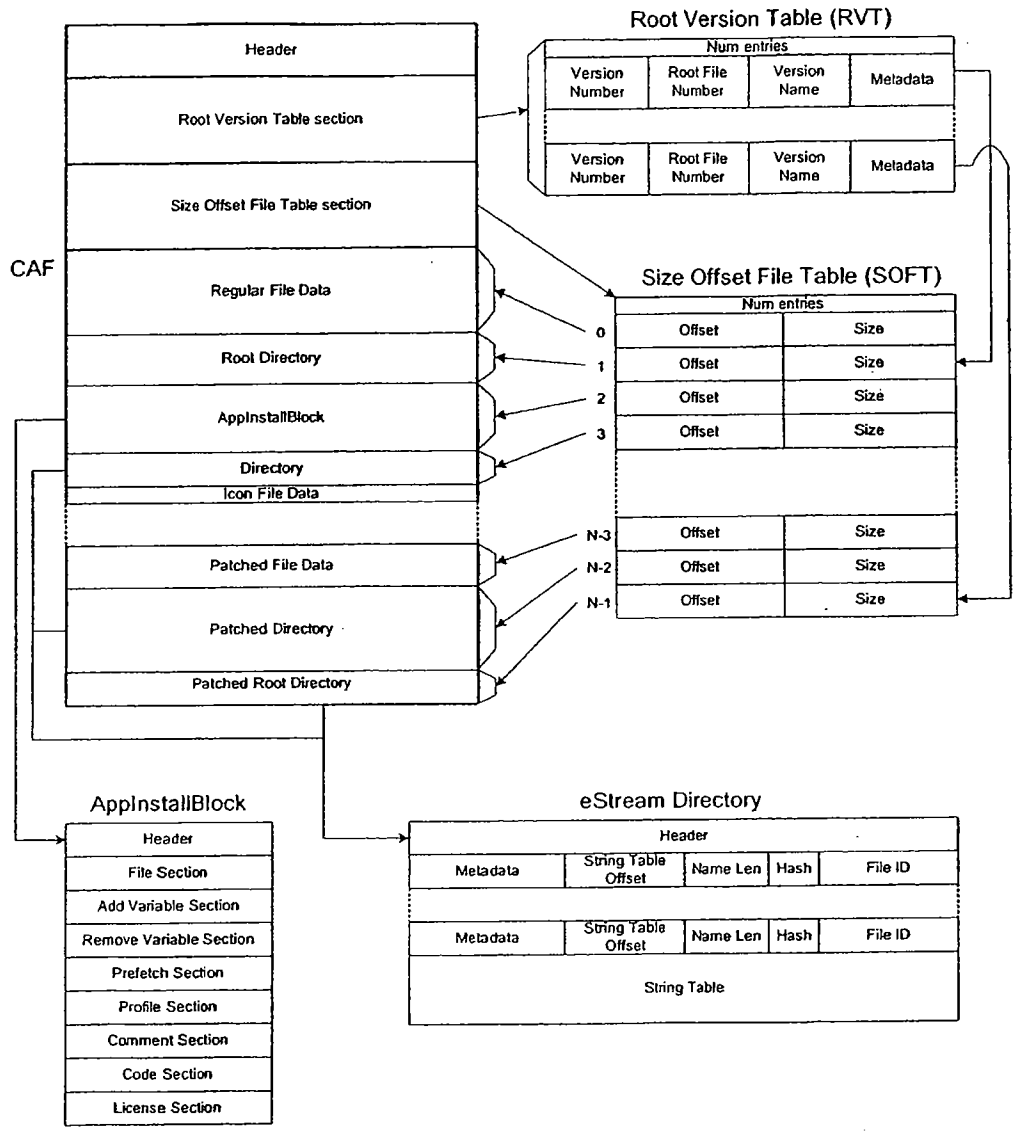
- **Format [1 byte]:** Format of this entry, should be 'l' for long filename format.

- **Index [1 byte]:** Number of this entry out of those used for this file's long name.
- **UNUSED [2 byte]:** Padding
- **NameHash [4 bytes]:** Hash value of the long file name. Algorithm TBD.
- **LongName [76 bytes]:** Long filename in Unicode format.

### *d. Icon files*

- **IconFileData [X bytes]:** Content of an icon file.

Format of the eStream Set

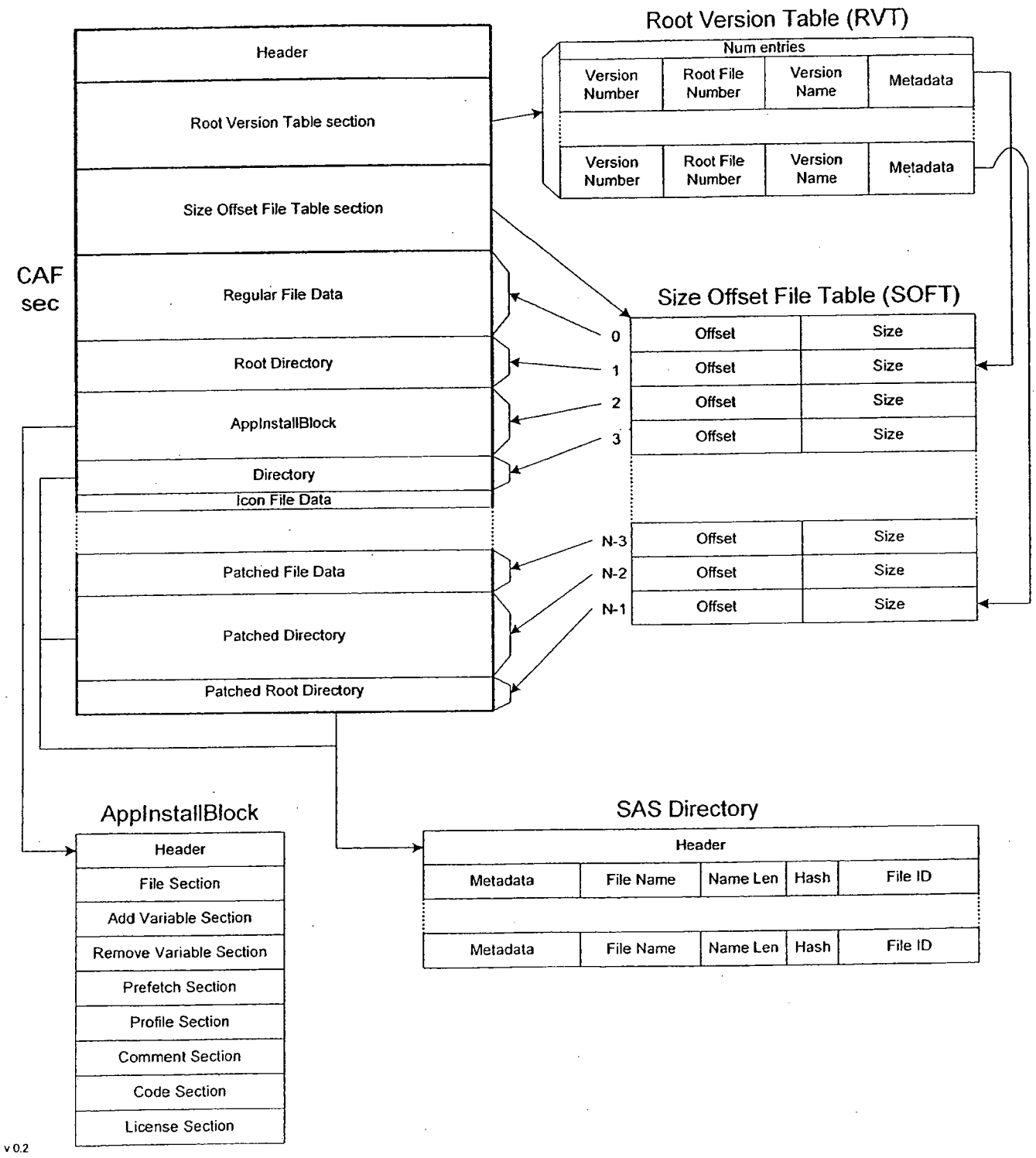



v 0.2

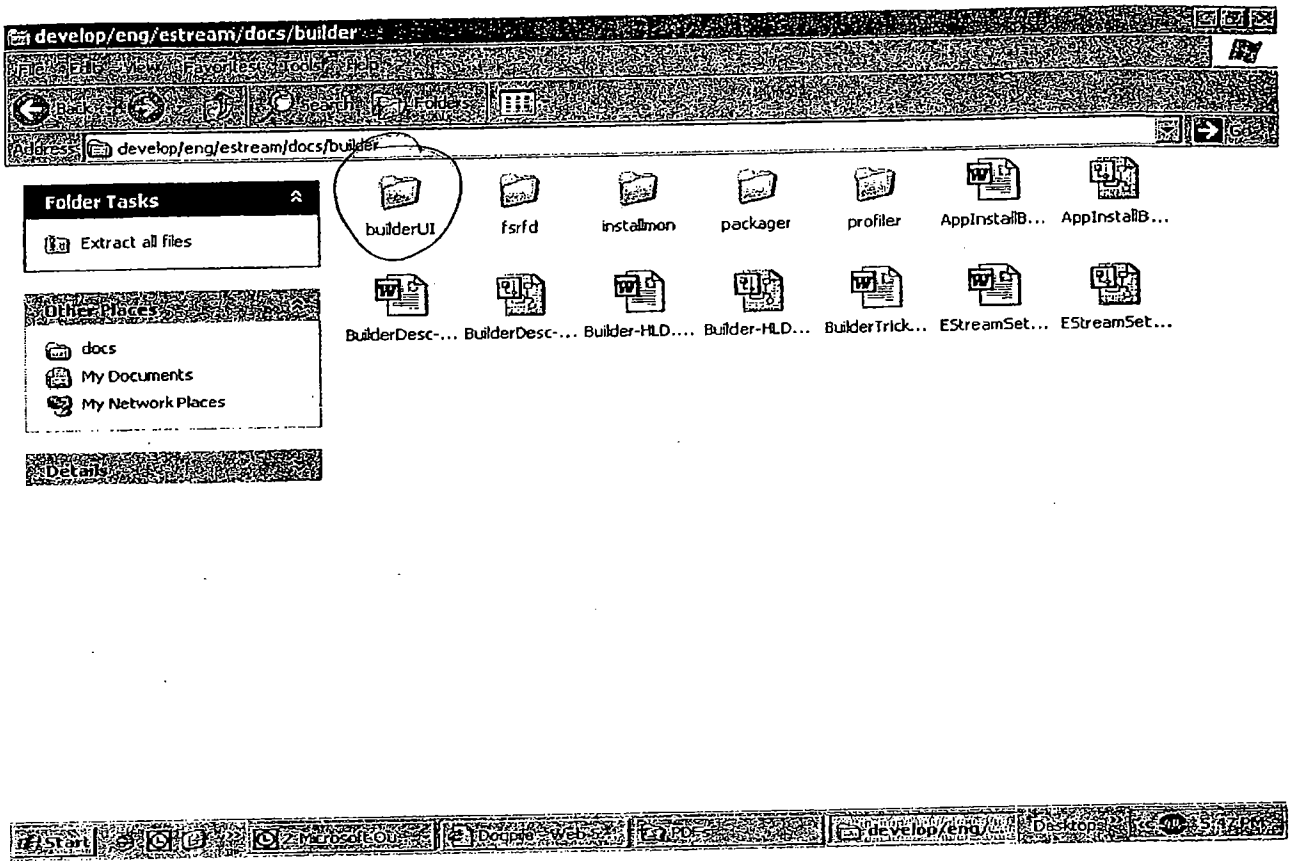
# Open Issues

- Where is the metadata associated with the Root directory located? Currently, root metadata is located in the root version table. All other files and directory metadata can be found in their parent directory.

# Content of eStream Set




v 0.2


develop/eng/estream/docs/builder
File Edit View Favorites Tools Help
Back Search Folders
Address develop/eng/estream/docs/builder
Folder Tasks
Extract all files
Other Places
docs
My Documents
My Network Places
Details
builderUI
fsrfd
installmon
packager
profiler
AppInstallB...
AppInstallB...
BuilderDesc-...
BuilderDesc-...
Builder-HLD....
Builder-HLD...
BuilderTrick...
EStreamSet...
EStreamSet...


start
develop/eng/...
Desktop

[/STANDARD.01]

# eStream BuilderUI Low Level Design

*Sanjay M Pujare*
*<Date>*

## Functionality

The BuilderUI is the user interface part of the Builder. The operator uses this interface to use various functions provided by the Builder. Note that this UI may or may not be a graphical user interface. This low-level design is based on the assumption that a graphical user interface is not necessary.

## Data type definitions

## Interface definitions

## Component Design

When the Builder UI is invoked with command line arguments which indicate that this was invoked by the Runonce mechanism of Windows, the control is transferred to the `InstallMon::startCaptureAfterReboot()` function with the command line arguments passed as arguments to the function. When the Builder is invoked normally, it presents a menu which is managed by the function `MainMenu`.

**MainMenu**

```
This function manages the following menu hierarchy. Each
menu option (leaf node) is followed by a function name in
parentheses that is called to handle the option.
1) eStream Set Menu
   1) New eStream Set (NewEStreamSet)
   2) Open eStream Set (OpenEStreamSet)
   3) Save New/Upgraded eStream Set (EstreamSetCreation)
2) Monitoring Menu
   1) Start Monitor (StartMonitor)
   2) Stop Monitor (StopMonitor)
   3) Check Status (CheckStatus)
   4) Inform Machine Reboot (InformMachineReboot)
   5) Get and Resolve Registry Set (GetRegistrySet)
   6) Get and Resolve Files Set (GetFilesSet)
3) Profiling Menu
   1) Set the location of app executable (GetAppPath)
   2) Gather Initial Profile (GatherInitialProfile)
4) eStream Set Creation Menu
   1) Set custom DLL (GetCustomDLL)
   2) Set User Comment (GetUserComment)
   3) Set environment variables (GetEnvVars)
```

4) Set Reboot flag (GetRebootFlag).
5) Set License Agreement (GetLicenseAgreement).

**NewEStreamSet**

{
If there is an existing eStream set that hasn't been saved, warn the user.
Get the following values from the user:

- Name of the app setup program in gAppSetup
- Dest directory where app will be installed in gDest-Dir (provide a default value).
- Dest location to store the new eStream set in gDestEstreamPath (provide a default value).

}

**OpenEStreamSet**

{
If there is an existing eStream set that hasn't been saved, warn the user.
Get the following values from the user:

- Location of the existing eStream set in gSrcEstream-Path (provide a default value).
- Whether the user wants to create an upgrade from this, or just wants to change the existing eStream set (gUp-grade)
- If this is an upgrade (gUpgrade is true), get all the values obtained by NewEstreamSet (i.e. gAppSetup, gDestDir, gDestEstreamPath).

Read the eStream set pointed to by gSrcEstreamPath;
Load the existing file tables in gSrcCopiedFiles, gSrcSpoofedFiles and gSrcEFSFiles arrays;
}

**EstreamSetCreation**

{
If there is no working eStream set, give error and re-turn.
if (gUpgrade) {
Call UpgradeEStreamSet() with appropriate arguments;
}
else {
Call CreateEStreamSet() with appropriate arguments;
}
if (gDestEstreamPath is not set) {
assert(this is an update of an existing eStream set

```
        and not an upgrade or a new eStream set creation);
      gDestEstreamPath = gSrcEstreamPath;
   }
   Save the eStream Set from memory to file to
      gDestEstreamPath;
}
```

**StartMonitor**

```
{
   If there is no working eStream set, give error and re-
   turn.
   if (gUpgrade) {
      Combine the gSrcSpoofedFiles and gSrcEFSFiles into an
      array fileTableArray as expected by startCapture
      below;
   }
   Call InstallMon::startCapture(gAppSetup, gDestDir,
        gUpgrade, fileTableArray);
}
```

**StopMonitor**

```
{
   If monitoring wasn't started, give error and return.
   Call InstallMon::stopCapture();
}
```

**CheckStatus**

```
{
   Ensure that monitoring was started and not stopped
   Call InstallMon::checkSetupStatus();
}
```

**InformMachineReboot**

```
{
   Ensure that we are in the middle of monitoring.
   Call InstallMon::machineToBeRebooted();
}
```

**GetRegistrySet**

```
{
   Call InstallMon::getRegistryList();
   Store the set in gNewRegistry data structure;
}
```

**GetFilesSet**

```
{
   Call InstallMon::getFilesList();
```

```
   Store the set in gNewFiles data structure;
   Also we need to capture changes made to INI files; since
   this has not been taken care of in the app install block
   and other parts of the Builder+Client we will need to
   make changes in all those components which are affected.
}
```

**GetAppPath**

```
{
   Ensure that all the InstallMon related data was captured.
   Get the location of the app that needs to be run
   to gather profiling info;
   Store it in gProfileAppPath;
}
```

**GatherInitialProfile**

```
{
   // this is the function that is used to get the initial
   // profile data (i.e. set of pages prefetched when an
   // eStream app is started for the first time)
   // implementation of this is yet to be defined
}
```

**GetCustomDLL**

```
{
   Get the location of the custom DLL file, validate that it
   is a DLL and store the path in gCustomDLLPath;
}
```

**GetUserComment**

```
{
   Get the user comment (optionally by browsing a text file)
   and store it in gUserComment;
}
```

**GetEnvVars**

```
{
   Call the InstallMon::setEnvVars() function;
}
```

**GetRebootFlag**

```
{
   Until we come up with an algorithm to determine if a re-
   boot is required for an eStream app, get this value from
   the user. The default is FALSE: we do not want to reboot
   the client PC when the user subscribes to this eStream
   app.
```

}

**GetLicenseAgreement**

```
{
   Get license agreement from the user. This could be:
      o either, a default OmniShift license agreement
      o or, a default ASP agreement
      o or, license agreement that the app displayed.
   Let the user decide and enter the proper one. Provide a
   default based on our policy.
}
```

Interesting issues to deal with:

# Testing design

## Unit testing plans

Testing of the UI itself is a comparatively trivial task. The testing will basically consist of traversing the whole menu hierarchy. Since the menu is similar to a typical File Open -> File Edit -> File Save kind of a user application, this can be tested using simple hooks.

## Stress testing plans

Since the Builder will be used in house at least initially only simple stress testing should be necessary. Make sure that Builder doesn't crash in the middle of processing so that we don't lose important data. Performance is not considered to be important.

## Coverage testing plans

Basically the following 3 paths will be exercised:

1. Create a new eStream set
2. Open an existing eStream set to modify some data in it
3. Open an existing eStream set to create an upgrade for it

## Cross-component testing plans

Will be tested as a component of the whole builder.

# Upgrading/Supportability/Deployment design

Deployment: This will be used in-house, so no deployment considerations.

# Open Issues

- We need to think about the problem of converting absolute file paths discovered in the monitoring process to paths relative to some application or system registry key. Although most cases may not present a problem, we may have some difficult cases, which may make this problem non-automatable i.e. we may need some user intervention. Consider the case:

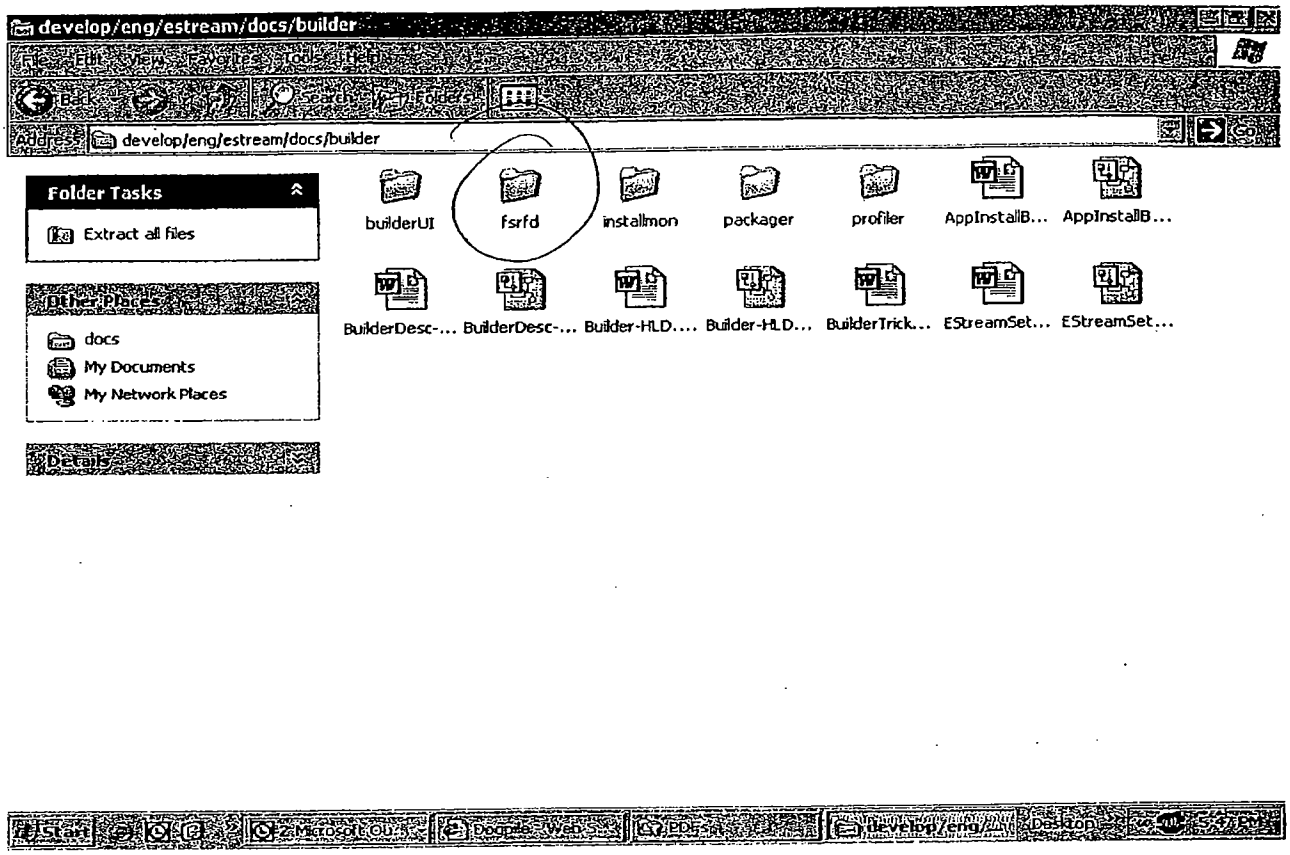
  KEYONE = C:\FOO\BAR

  KEYTWO = C:\FOO

  KEYTHREE = BAR

  If we notice that a file was copied to C:\FOO\BAR it won't be possible to convert this absolute path to a unique relative path since there are 2 solutions possible: %KEYONE% or %KEYTWO%\%KEYTHREE%.

  The way to solve this is by tracking only a set of well-known environment variables and registry keys. Also in this set we prioritize all of them. So in the above case, %KEYONE% will be preferred over %KEYONE%\%KEYTHREE% just because of the way they were prioritized.


develop/eng/estream/docs/builder
File Edit View Favorites Tools Help
Back Search Folders
Address develop/eng/estream/docs/builder
Folder Tasks
Extract all files
Other Places
docs
My Documents
My Network Places
Details
builderUI
fsrfd
installmon
packager
profiler
AppInstallB...
AppInstallB...
BuilderDesc-...
BuilderDesc-...
Builder-HLD....
Builder-HLD...
BuilderTrick...
EStreamSet...
EStreamSet...


Start
Microsoft Out...
Google - Web...
develop/eng/...
Desktop

# eStream FSRFD Low Level Design

*Sanjay M Pujare*
[redacted]

## Functionality

The File System and Registry Filter Driver (FSRFD) is a part of the Builder module that monitors file system and registry updates initiated by the Builder process. This driver just intercepts such requests and records them and returns the recorded data to the Install Monitor (*INSTALLMON* described in another LLD) program when requested by the latter. All the intelligence, such as any decision-making logic, resides in the INSTALLMON.

For registry updates such as add or modify, the FSRFD needs to record the value added or modified. For registry deletes only the value name needs to be recorded. For file updates, there is no need to record the file contents added or modified, since the Builder would be interested only in the final contents of a file.

**Note:**

1. **This does not cover those rare cases where an existing file is updated by an application install, and the eStream client would need to make the same updates. This kind of functionality is difficult to implement and will not be considered for 1.0.**
2. **The FSRFD will be used to monitor only one install at a time to simplify the design of the FSRFD. That means you cannot invoke multiple instances of the Builder at a time to monitor multiple installations. All Builder invocations on a machine have to be strictly sequential.**
3. **This design is based on the driver model for WinNT and Win2K. The Win98 driver is not covered here (yet).**

## Data type definitions

The following struct is used to communicate information related to activating the FSRFD. Specifically, the process-id of the INSTALLMON and the 2 drives whose accesses need to be monitored are passed.

```
struct MonitorActivate_t {
   ULONG processId; // PID of INSTALLMON
   UCHAR sysDrive;  // System drive letter
   UCHAR destDrive; // Dest drive letter
};
```

The following struct is used to return monitored data back to the INSTALLMON. Note that this is a variable size struct where the last field keyName is an array of one wide-char, but in reality is an array of length whose value is the sum of 3 length fields in the struct (nameLength, valueNameLength and dataLength).

```
struct IMON_ENTRY {
  UCHAR regOrFile; // 'R' for registry, 'F' for files
                   // and 'E' for end of data
  UCHAR updateType; // 'A' for add, 'D' for delete,
                   // 'U' for update
  UCHAR valueType;  // for registry only: value type
// REG_SZ, REG_DWORD, REG_BINARY,
// REG_DWORD_LITTLE_ENDIAN, REG_DWORD_BIG_ENDIAN,
// REG_EXPAND_SZ, REG_LINK, REG_MULTI_SZ, REG_NONE,
// REG_QWORD, REG_QWORD_LITTLE_ENDIAN,
// REG_RESOURCE_LIST
  ULONG nameLength; // length of name (file or
                   // registry key) in wchars
  ULONG valueNameLength; // length of value name (if
                         // it exists) in wchars
  ULONG dataLength;     // length of data in bytes
  WCHAR keyName[1];     // keyName followed by
                   // valueName followed by
                   // data: note none of these are
                   // null terminated & are wide
                   // chars
};
```

Note about the updateType field: 'A' is used for file creation and 'U' is used for any updates to the file. So if a 'U' is seen without an 'A' for a file that means the file was modified but not created in this session.

The following struct is used as the device extension in the FSRFD devices. Note that this extension is used for all device objects: the device that is created in the DriverEntry function to represent an "INSTALLMON" device for the INSTALLMON to access our driver as well as the devices that are created to create a filter layer above existing drives.

```
enum DEVICE_TYPE {
          INSTALLMONINTERFACE,
          STANDARD
};
```

```
struct DeviceExtension_t {
          PDEVICE_OBJECT deviceObject; // device
                         // for lower layer
          DEVICE_TYPE type; // see design
          KMUTEX pDeviceMutex;
          ProcessIdList
          // pointer to list of process-ids we are
          // interested in monitoring
          EntryList
          // This is the list that stores all the
          // info captured by the FSRFD until each
          // entry is queried by INSTALLMON
};
```

There is an array for FastIo that stores all the FastIo function pointers which is required in a file system filter driver such as this. Note that we need to provide entry points for all (or most?) of the FastIo routines in the dispatch table, since we need to pass the request down to the lower layer driver even if we are not interested in intercepting the request. For only some of the requests (e.g. all the FASTIO_*WRITE* requests), we would be recording the file access and creating an entry to be returned to INSTALLMON. The Component Design section describes in more detail the FastIO routines that are implemented by this driver.

# Interface definitions

## INSTALLMON interfaces

Since the FSRFD is a driver, it cannot provide directly callable APIs. Instead the INSTALLMON communicates with the FSRFD using the DeviceIoControl Win32 API. It uses Ioctl codes MON_ACTIVATE, MON_DEACTIVATE and MON_GET_ENTRY. The DeviceIoControl API looks as follows:

```
BOOL DeviceIoControl(
  HANDLE hDevice,              // handle to device
  DWORD dwIoControlCode,       // operation control code
  LPVOID lpInBuffer,           // input data buffer
  DWORD nInBufferSize,         // size of input data buffer
  LPVOID lpOutBuffer,          // output data buffer
  DWORD nOutBufferSize,        // size of output data buffer
  LPDWORD lpBytesReturned,     // byte count
  LPOVERLAPPED lpOverlapped   // overlapped information
);
```

The semantics of each of the MON_* codes is defined below. Note that this is described from the caller's (i.e. INSTALLMON) point of view.

## Ioctl 1 – MON_ACTIVATE

Input:

hDevice:
is the handle to the FSRFD device created.

dwIoControlCode:
The value MON_ACTIVATE

lpInBuffer:
Address of MonitorActivate_t struct. This contains the process id of INSTALLMON.

nInBufferSize:
Sizeof(MonitorActivate_t)

Output:

lpOutBuffer:
Should be a ptr to a ULONG (at least).

nOutBufferSize:
Size of above buffer

lpBytesReturned:
Should be a ptr to a DWORD where byte count of data returned in lpOutBuffer is returned.

Comments:

MON_ACTIVATE is sent to the FSRFD when the INSTALLMON wants to start monitoring an installation. MON_ACTIVATE can be sent only when the FSRFD is not already active - either after the driver is loaded the first time or after the last MON_DEACTIVATE request.

Errors:

- STATUS_ALREADY_ACTIVE: FSRFD was already activated. The processId of the old activation is returned in the ULONG pointed by lpOutBuffer.
- STATUS_INVALID_ARG: One of the arguments passed is not valid (either invalid MonitorActivate_t ptr or processId).

- STATUS_INVALID_DRIVE: Either the sysDrive or the destDrive (or both) is invalid.

## Ioctl 2 – MON_DEACTIVATE

Input:

hDevice:
is the handle to the FSRFD device created.

dwIoControlCode:
The value MON_DEACTIVATE

lpInBuffer:
NULL, or pointer to a ULONG where the ULONG has a non-zero value indicating a "forced" deactivation. A "forced" deactivation is done when the FSRFD still has entries that are not going to be retrieved.

nInBufferSize:
0 or size of ULONG (depending on the above).

Output:

No need for OUT arguments.

Comments:

MON_DEACTIVATE is sent to the FSRFD when the INSTALLMON wants to stop monitoring an installation. MON_DEACTIVATE can be sent only when the FSRFD is already active i.e. after the last MON_ACTIVATE request.

Errors:

- STATUS_NOT_ACTIVE: FSRFD was already deactivated. No special action is needed to be taken for this error condition. The caller can simply send a new MON_ACTIVATE.
- STATUS_PENDING_DATA: The FSRFD has some entries that were not read using

MON_GET_ENTRY. This error is only returned in case of non-forced deactivation.

## Ioctl 3 – MON_GET_ENTRY

Input:

hDevice:
is the handle to the FSRFD device created.

dwIoControlCode:
The value MON_GET_ENTRY

lpInBuffer:
NULL.

nInBufferSize:
0

Output:

lpOutBuffer:
Should be a ptr to a IMON_ENTRY struct.

nOutBufferSize:
Size of above buffer

lpBytesReturned:
Should be a ptr to a DWORD where byte count of data returned in lpOutBuffer is returned.

Comments:

MON_GET_ENTRY is sent to get the next "entry" from the FSRFD. An "entry" is a record of a registry or file update intercepted by the FSRFD. The details are returned in the IMON_ENTRY struct passed in the lpOutBuffer argument. Note that the FSRFD uses an event object (created using the IoCreateNotificationEvent API) to signal the INSTALLMON that an "entry" is available to be read. INSTALLMON waits on this event object before retrieving the entry using MON_GET_ENTRY.

Errors:

- STATUS_NOT_ACTIVE: FSRFD was not activated.
- STATUS_INVALID_ARG: One of the pointer arguments passed is not valid.
- STATUS_INSUFF_BUFFER: The buffer size indicated by nOutBufferSize is insufficient for the current "entry" data.

## Ioctl4 – MON_GET_ERROR

We need this to indicate occurrence of an error whenever this occurs in any of the dispatch functions. We can either implement this control code or just return an error code for any MON_GET_ENTRY call that reflects that an error occurred.

## Event Object interface

As mentioned above, an event object (lets call it IMON event object) will be used to signal the INSTALLMON that a new entry is available. This event object will be created in the FSRFD in the processing of MON_ACTIVATE ioctl, as:

```
ext->pEvent = IoCreateNotificationEvent(
   L"\\BaseNamedObjects\\INSTALLMONEVENT",
         ext->eventHandle);
```

Whenever the FSRFD has a new entry, it signals using the above event object as follows:

```
KeSetEvent(ext->pEvent, 0, FALSE);
```

Whenever INSTALLMON gets the next entry using MON_GET_ENTRY ioctl, the FSRFD resets the event (if the list of entries is going to be empty after this entry) as follows:

```
KeClearEvent(ext->pEvent);
```

Whenever a MON_DEACTIVATE is processed, the FSRFD will close the event as:

```
ZwClose(ext->eventHandle);
```

# Kernel or Low Level Driver Interfaces

Every driver needs a `DriverEntry` routine that is called when the driver is first loaded. This routine for FSRFD is described in the Component Design section.

The FSRFD inserts "hook routines" that intercept relevant registry and file system calls. The Component Design section describes which hook routines are inserted and what they do.

# Component Design

### *Global variables*

#### pImonDevice

This global variable points to the device object created in the DriverEntry function for the "\\Device\\installmon" device.

#### DriverEntry

```
NTSTATUS DriverEntry(IN DriverObject, IN RegistryPath)
{
          IoCreateDevice for "\\Device\\installmon";
          pImonDevice = pDeviceObject returned above;
          ext = pImonDevice->DeviceExtension;
          ext->type = INSTALLMONINTERFACE;
          IoCreateSymbolicLink with
              "\\DosDevices\\installmon";
          for all IRP_MJ_* values upto
               IRP_MJ_MAXIMUM_FUNCTION {
             DriverObject->MajorFunction[IRP_MJ_*] =
                  ImonDispatch
          }
          Setup the unload driver function
          DriverObject->FastIoDispatch = address of
             our fast io dispatch table;
             Note that we are interested only in
             FAST_IO_*WRITE* routines for getting our
             entries, however we need to implement all
             of them to call lower layered drivers. All
             our FastIo funcs are called ImonFastIo*;
          Create the necessary mutexes;
          Use PsSetCreateProcessNotifyRoutine to set a
          process create callback routine
          ImonProcessCallback;
}
```

#### ImonProcessCallback

```
    {
          // similar to ImonProcessCallback
          // This function is called every time a
```

```
                // process on the system is created or
                // deleted. We need to figure out (by
                // checking the parent id in our list)
                // if we need to add or remove this process
                // from our list
                lock the mutex for ProcessIdList
                if not activated just return;
                if (this is process create) {
                   Look up the parent process id in the
                   ProcessIdList
                   If present, add this process id to the
                      ProcessIdList
                }
                else {
                   if this process id is present then
                      remove the process id
                }
                release the mutex
        }
```

**ImonDispatch**

```
        {
                // similar to FilemonDispatch
                // Instead of registering a different
                // function for each IRP_MJ_* this function
                // is called for all of them and this one
                // dispatches the right on based on the
                // control code
                This gets called for all IRP_MJ_*;
                if the device extension type tells us
                   INSTALLMONINTERFACE
                      call ImonDeviceFunc
                else
                      call ImonHookFunc
        }
```

**ImonDeviceFunc**

```
        {
                // similar to FilemonDeviceRoutine
                // This function is called whenever an Ioctl
                // comes from the Installmon process that is
                // meant to be a command for this FSRFD.
                This is a request from the INSTALLMON using
                the INSTALLMON device. We would mainly be
                processing IRP_MJ_DEVICE_CONTROL, although
                ImonFastIoDeviceControl should have been
                called. So if we come here just call
```

```
        ImonFastIoDeviceControl
        Call IoCompleteRequest?
}
```

**ImonHookFunc**

```
{
        // similar to FilemonHookRoutine
        // This is the hook function that is called
        // for all the I/O requests that is made by
        // the I/O manager that need to go through
        // this driver (i.e. for those requests
        // that we are filtering).
        We are interested in recording:
           IRP_MJ_CREATE where the Irp->
              Parameters.Create.Options indicates
              a new file create (as opposed to an
              existing file open)
           IRP_MJ_WRITE
        Note that we have to get the current
        process-id using PsGetCurrentProcessId and
        look it up in the ProcessIdList (note: you
        have to exclude the first process-id since
        that belongs to the Installmon and not the
        setup program) and only if that search is
        successful, record the entry
        Note that we need to get the filename from
           the FileObject using code similar to
           FilemonGetFullpath
        Also we should be recording the entry when
        the request is successfully completed. So do
        this in the completion routine in a manner
        similar to filemon.
        To record:
           create a relevant IMON_ENTRY record;
           Call ImonAddEntry with this record;
        Pass all Irps to lower layer driver using
           IoCallDriver and getting the lower layer
           device object from this device's
           ext->deviceObject
}
```

**ImonFastIoDeviceControl**

```
{
        // similar to both
        // FilemonFastIoDeviceControl and
        // ImonDispatchIoctl
        // This function gets called for all the
```

```
                // IOCTLs. If it is from Installmon, we need
                // to process the MON_* commands or else
                // just pass on the command to the lower
                // layer driver.
                Get the current device's extension
                if type indicates INSTALLMONINTERFACE {
                   switch (IoControlCode) {
                      case MON_ACTIVATE:
                         call ImonActivate;
                         break;
                      case MON_DEACTIVATE:
                         Call ImonDeactivate;
                         break;
                      case MON_GETENTRY:
                         Call ImonGetEntry;
                         break;
                   }
                }
                else {
                   pass it down using deviceObject and the
                   fastio hook for Ioctl
                }
        }
```

**ImonAddEntry**

```
        {
                // similar to ImonEnqueueRegEntry and
                // createRegEntry etc.
                // This function adds a IMON_ENTRY node
                // to our list: this list is eventually
                // returned to InstallMon
                create an entry rec from nonPagedPool
                Grab a mutex to modify EntryList
                Add the entry rec to EntryList
                release the mutex
                KeSetEvent for IMON event object
        }
```

**ImonActivate**

```
        {
                // similar to ImonActivate and various
                // Filemon funcs called by
                // FilemonFastIoDeviceControl
                // This is called by our own func that
                // handles IOCTLs when a MON_ACTIVATE is
                // sent by InstallMon
                Look at the ProcessIdList to ensure that we
```

```
                are not already active (1st process id).
                If we are, return with an error with that
                process id
                Grab a mutex
                Add a node to ProcessIdList with the
                processId;
                Release the mutex;
                Create the IMON event object;
                HookDrive(sysDrive);
                HookDrive(destDrive);
                HookRegistry();
    }
```

**ImonDeactivate**

```
    {
                // Same as above except for MON_DEACTIVATE
                If we are already deactivated
                   return with an error;
                If EntryList is not empty and this is not
                   a forced deactivation then
                   return with error;
                Clear and Delete the IMON event object;
                Free the ProcessIdList;
                Free the EntryList;
                UnhookRegistry();
    }
```

**ImonGetEntry**

```
    {
                // When the InstallMon sends a MON_GET_ENTRY
                // this function is called.
                Grab the mutex to access EntryList
                get next entry and remove from the
                list;
                if no entry then {
                   if (deactivated and first process in
                        the processIdList is dead) {
                      Make a new entry of "end of data"
                        type;
                   }
                   else {
                      there is some problem (should not
                        happen)
                   }
                }
                copy the entry into the OUT buffer
                release the mutex;
```

```
}
```

**HookDrive, UnhookDrive**

```
{
        // very similar to Filemon's HookDrive
        // Hence not described here
        // Similarly UnhookDrive
        // When a MON_ACTIVATE comes, we need to
        // make sure that all the requests to the
        // system drive or dest drive are filtered
        // through us. e.g. If "C:" is the system
        // drive, HookDrive will register this
        // as the filter driver for all IO for "C:"
        // Similary for UnhookDrive.
}
```

**HookRegistry, UnhookRegistry**

```
{
        //similar to Installmon's (Un)HookRegistry
        // We need to make sure that all Registry
        // functions are replaced, with our funcs
        // so that these funcs get called whenever
        // a process is trying to access the
        // registry. Our hook funcs will in turn
        // call the real funcs after queueing an
        // entry
}
```

**ImonFastIo* routines**

```
{
        // These are very similar to FilemonFastIo
        // routines except that we need to intercept
        // only FAST_IO_WRITE,
        // FAST_IO_MDL_WRITE_COMPLETE,
        // FAST_IO_WRITE_COMPRESSED,
        // FAST_IO_WRITE_COMPLETE_COMPRESSED
        Call the lower layer driver;
        if the request was successfully completed {
           Record the details into our entry rec
           and enqueue it similar to how ImonHookFunc
           does it;
        }
}
```

Interesting issues to deal with:

- Make sure that we use non-paged memory as required. e.g. all the nodes of the processIdList and entryList will be from non-paged pool. According to Bob, we do not need to always allocate non-paged memory – for Registry related nodes (i.e. when it is 'R' type IMON_ENTRY node) we can allocate paged memory. However this needs to be checked – since there will be pointers between these nodes, we need to make sure that this will work properly.
- We have to make sure that a 2-phase install works with this: some setups ask you to reboot the machine and after the reboot the setup continues. For the FSRFD we need to make sure that after the reboot the FSRFD is already loaded before the 2nd phase of the install starts. In the FSRFD we may need to make sure that when the "Runonce" registry key is updated (the installer is trying to do a 2-phase install and setting the 2nd phase exe as the value of "Runonce") we capure that info and accordingly co-ordinate with the Installmon to do the right thing. To ensure that this driver is started at the right time on start-up (since the Builder machine is going to be an in-house dedicated machine, it is okay), we need to add to the System registry (under `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services`) appropriate values for `Start`, `Group` and `Tag` (and possibly others) value names. Actually this is going to be implemented in InstallMon as a user initiated event in which case the user informs the Builder UI (which in turn informs the InstallMon) that a reboot is imminent. In that case the InstallMon can try to find all possible ways in which the setup.exe is achieving this:
  - The "RunOnce" key or one of its other incarnations (e.g. RunOnceService, RunOnceEx etc) has been modified. We need to figure out exactly which one of the "*RunOnce*" can be modified.
  - The setup.exe has actually added itself (or another exe) to the startup folder. If that is the case, we can do the same trick here: replace that entry with an entry pointing to the BuilderUI with the original setup.exe value as an argument to the BuilderUI.
- An issue that hasn't been resolved is any user interaction (and user input) that has taken place during installation that is being monitored. For example, an installation may ask for a port number that it may store in a registry key. The solution suggested is as follows: This has to be a manual process. The Builder user should record all the manual interaction and manual data input that has taken place. He should recreate the same interaction in the custom DLL that the appInstallBlock provides for that app. This custom DLL at eStream app subscribe time can do the same thing that was observed during original application install. Alternatively we can request the ISV's co-operation in doing this. (May be this bullet should be transferred to a different doc such as the InstallMon LLD.).
- There is another unresolved issue about handling h/w or s/w dependent things that the installer does: we will have to handle this case by case basis and any knowledge we gain as a result of this, we should consolidate in the Builder components. e.g. we may notice that some installations may depend on IE4 or IE5 being there. Of course, one of the pre-requisites of the Builder is that it will be run on a pristine machines, so that we capture a maximal installation when it is taking place.

# Testing design

## Unit testing plans

This will be unit tested using the INSTALLMON program (or its early prototype). The INSTALLMON program sends all the required Ioctls like MON_ACTIVATE, MON_DEACTIVATE and MON_GETENTRY. The INSTALLMON output will be used to check the correctness of the FSRFD. This means the INSTALLMON itself should be assumed to be correct.

In addition to the above, we actually need to write one or more test programs that exercise the FSRFD. These test programs will be run as if they were App Installers i.e. as child processes of the INSTALLMON. The test programs should be written to exercise:

- All file systems (FAT, FAT32, NTFS, HPFS, compressed drives and others), since the FastIo routines need to be tested.
- For each of the file systems:
  - File create (with and without the old file being there)
  - File update (existing file appended as well as modified in the middle)
  - File touch (e.g. get the version of a DLL file) – this is not yet covered by this design
- Registry updates:
  - create a key
  - delete a subkey
  - delete a named value from a key
  - RegReplaceKey (we need to investigate if this is broken down into smaller reg calls).
  - RegRestoreKey (same comment as above applies)
  - RegSetKeySecurity (we failed to address this in the design)
  - RegSetValueEx
  - RegLoadKey and RegUnLoadKey

## Stress testing plans

The FSRFD will be stress-tested using the above testing strategy by varying the rate at which files/registry are updated and under a variety of conditions (memory/disk, other processes).

## Coverage testing plans

The unit testing above also covers Coverage testing.

## Cross-component testing plans

This will be tested with the Installmon program which is enough for cross-component testing.

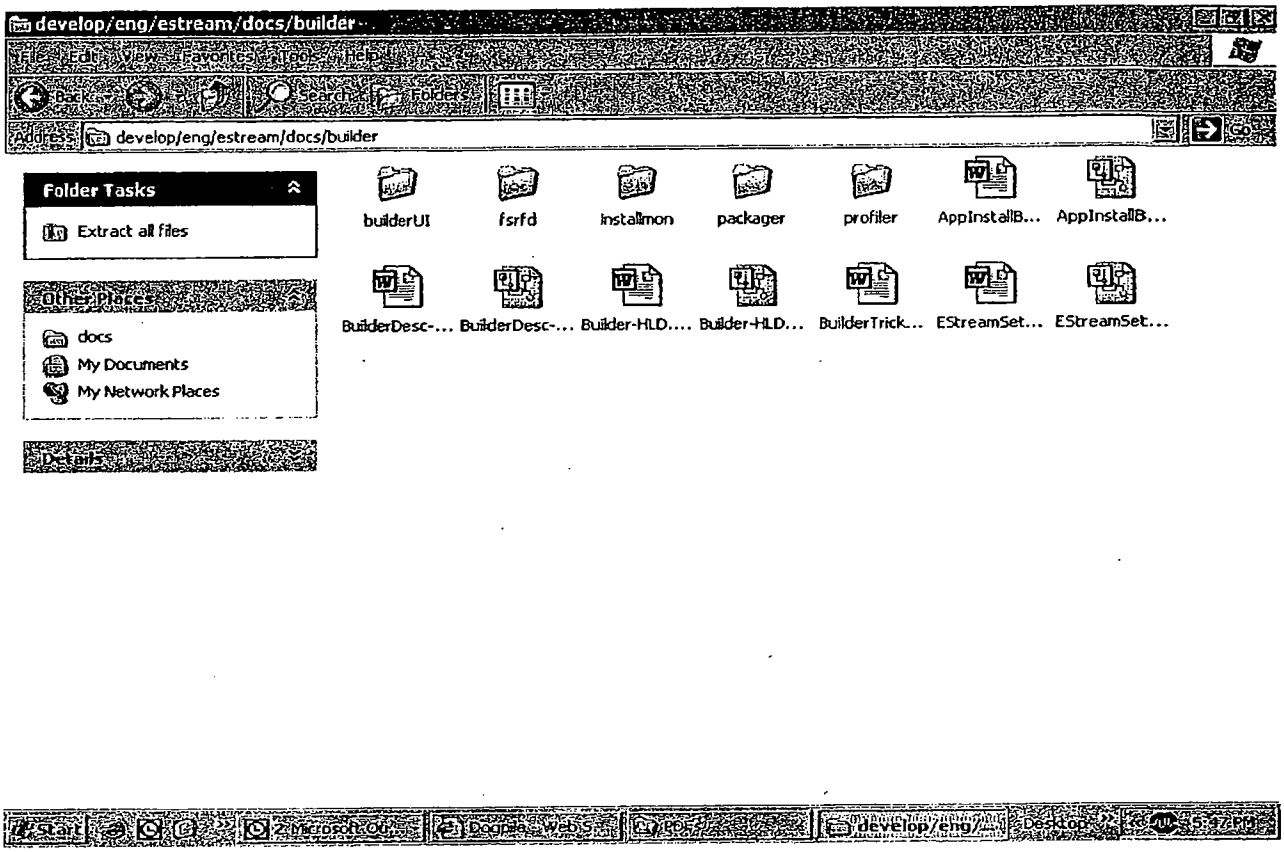
# Upgrading/Supportability/Deployment design

Deployment: This will be used in-house, so no deployment considerations.

# Open Issues

These issues are not necessarily FSRFD related, but are listed here just to remind ourselves.

- Do we need to worry about environment variables? It looks like most installations (and their apps) won't be looking at environment variables.
- What about the .lnk files (shortcuts). It looks like the client guys will have to change all the .lnk files based on the actual client's settings. This issue has been addressed either in the InstallMon or Packager. Pls see those documents. (The `IShellLink` interface has been suggested).
- Also what about when device drivers are installed? There is no impact on the builder but the client will need to reboot and hopefully the existing appInstall-Block and the custom initialization dll should be able to take care of it.


develop/eng/estream/docs/builder
File Edit View Favorites Tools Help
Back Search Folders
Address develop/eng/estream/docs/builder
Folder Tasks
Extract all files
Other Places
docs
My Documents
My Network Places
Details
builderUI
fsrfd
installmon
packager
profiler
AppInstallB...
AppInstallB...
BuilderDesc-...
BuilderDesc-...
Builder-HLD....
Builder-HLD...
BuilderTrick...
EStreamSet...
EStreamSet...


develop/eng/...
Desktop

# eStream Installmon Low Level Design

*Sanjay M Pujare*
*<Date>*

## Functionality

The Installmon is a part of the Builder module that talks to the FSRFD to monitor file system and registry updates initiated by the Builder process. The FSRFD driver just intercepts such requests and records them and returns the recorded data to Installmon when requested by the latter. All the intelligence, such as any decision-making logic, resides in the Installmon.

**Note:**

1. **This does not cover those rare cases where an existing file is updated by an application install, and the eStream client would need to make the same updates. This kind of functionality is difficult to implement and will not be considered for 1.0.**

2. **Somewhere in the docs (may be the user docs?) it should be mentioned that the Builder's (or rather Installmon's) job could be made easier by the user by following these guidelines:**

   - **Make sure that values in all the registry keys are as distinct as possible. We may need to create a special Win2K or WinNT installation where each key (or ValueName within a key) will be created with a distinct or unique value as far as possible. E.g. If the default Windows installation creates 2 keys FOO and BAR and stores the same value "C:\Windows\System32" in both of them, we wouldn't know which one of those keys is used when a file is copied to "C:\Windows\System32". To solve this problem, all the effort should be made to ensure that FOO and BAR have distinct values.**

   - **When the Builder operator is installing an app under the Installmon, she should also make sure that the install script is given a distinct or unique value for each of the user inputs that may be used to set a registry key or an environment variable. This is especially necessary when the inputs seem to be totally unrelated. For example, vendor name and installation directory name. If both are entered as "Microsoft" then that could cause confusion to the Installmon.**

3. **There are 2 ways in which registry changes and file system changes can be captured:**

   - **using a kernel mode driver such as the FSRFD, Or**

   - **using a diff'ing mechanism for both the registry as well as the file system.**